Programación y personalización SIG

Albert Gavarró Rodríguez

PID_00174018

Material docente de la UOC

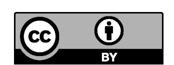


Albert Gavarró Rodríguez

Ingeniero en Informática
por la Universidad Rovira i Virgili.
Consultor de los Estudios
de Informática, Multimedia
y Telecomunicación
de la Universitat Oberta
de Catalunya.

Primera edición: febrero 2011

Av. Tibidabo, 39-43, 08035 Barcelona
Diseño: Manel Andreu
Material realizado por: Eureca Media, SL
Depósito legal: B-5.574-2011



Introducción a la programación orientada a objetos

Albert Gavarró Rodríguez

PID_00174495



Índice

Introducción

Muchos de nosotros utilizamos habitualmente los ordenadores y estamos familiarizados con el concepto *programa*. Dentro del ordenador hay muchos de ellos, y nos ofrecen multitud de funcionalidades: escribir documentos, hacer cálculos, e incluso jugar. Pero, ¿qué es un programa?

Aunque no lo creamos, hay programas en todas partes. Basta con que nos fijemos en una lavadora. Seguro que tiene un programa para la ropa delicada, otro para el algodón e incluso uno que sólo centrifuga. ¿Son iguales estos programas y los del ordenador? Pues, a grandes rasgos, sí. Veamos un par de ejemplos más: las notas que tenemos en nuestras agendas y una receta de cocina también son programas. Sorprendente, ¿no? Mucho más sorprendente es el hecho de que nosotros mismos ejecutamos programas de la misma forma que los ejecuta un ordenador.

Un programa no es más que un conjunto de instrucciones que permiten llevar a cabo una tarea. En el caso de la lavadora, el programador nos ofrece distintos programas entre los que podemos elegir. De nuestra elección, es decir, del programa, depende el conjunto de acciones que llevará a cabo la lavadora una vez encendida. Así, la lavadora puede desempeñar distintas tareas (lavado corto, lavado largo, etc.) que están perfectamente definidas por los distintos programas que nos ofrece. Las notas de nuestra agenda y la receta de cocina, al igual que un programa, también son un conjunto de instrucciones que hay que realizar para llevar a cabo las tareas "conservar el empleo" y "cocinar un plato". En ambos casos, son las indicaciones escritas las que controlan, de forma más o menos precisa, las acciones que llevamos a cabo. Son programas escritos para que nosotros mismos los ejecutemos.

Estructura de un programa

Un programa (sea informático o no) está compuesto por cuatro partes bien diferenciadas:

- **Código.** Es el conjunto de instrucciones en sí. Normalmente, el código está escrito de forma que sea fácil de entender y manipular por una persona.

- **Memoria.** Ofrece un espacio al programa para almacenar datos y recuperarlos más tarde.

- **Entrada.** Es el conjunto de datos que el programa recibe mientras se ejecuta y que condicionan las acciones que éste realiza y, en consecuencia, los

resultados que genera. Normalmente, los datos proceden del usuario (pulsaciones del teclado, movimientos y pulsaciones del ratón, etc.), pero también pueden venir de otros programas. En este último caso tenemos, por ejemplo, los datos enviados por un servidor web a nuestro navegador.

- **Salida.** Es el conjunto de datos generados en forma de resultado durante la ejecución del programa. Estos datos pueden percibirse como acciones desde el punto de vista del usuario. Los resultados pueden ser, pues, variopintos: un número, una hoja de papel impresa, una imagen en la pantalla, etc.

Volvamos al ejemplo de la lavadora: el código son las distintas acciones que ésta lleva a cabo, que dependen, ni más ni menos, del programa elegido y de las entradas (la cantidad de detergente, suavizante y lejía, y la ropa sucia). La salida, en este caso, está clara: la ropa lavada. Que esté más o menos limpia dependerá del tipo de ropa, de cómo estaba de sucia, del programa que hayamos escogido y de la cantidad de producto que hayamos usado, en fin, de las entradas. Por otro lado, la lavadora también usa la memoria para saber en qué punto está dentro del programa seleccionado, el tiempo que le queda para pasar al siguiente estadio de lavado, etc.

Lenguajes y compiladores

Más arriba hemos mencionado que los programas están escritos de forma que resulten fáciles de entender y manipular por una persona. Sin embargo, no siempre fue así. En los albores de la informática, los programas se escribían en forma de agujeros en unas tarjetas especiales. Con estos agujeros se representaban los datos y el código. Pero pasar de la voluntad del programador a los agujeros y viceversa no era tarea fácil. Afortunadamente, años más tarde aparecieron los denominados lenguajes de programación, que utilizaban una sintaxis muy parecida a la de un lenguaje natural para describir las acciones que debía realizar el ordenador. Estos programas eran interpretados posteriormente por otro programa llamado compilador, que se encargaba de traducir todas esas sentencias a algo más liviano para el ordenador. Esta forma de programar, con pocos cambios, es la que ha llegado a nuestros días.

El lenguaje Java

En las próximas páginas daremos las nociones básicas para empezar a programar un ordenador. Si bien estos fundamentos nos permitirán abordar satisfactoriamente casi cualquier lenguaje de programación, este tutorial utiliza el lenguaje Java como herramienta de aprendizaje. Las razones para escoger Java son diversas:

- Es un lenguaje de los llamados "de alto nivel", lo que significa que está más cerca del lenguaje natural que otros.

- Está ampliamente difundido: en móviles, PDA, páginas web, ordenadores...
- Es multiplataforma, es decir, un programa escrito en Java puede ejecutarse en casi cualquier máquina.
- Es un lenguaje orientado a objetos, que es quizás el paradigma de programación más cercano al razonamiento humano.

Objetivos

Una vez finalizado el módulo, deberíamos ser capaces de:

1. Entender pequeños programas escritos en Java o en cualquier otro lenguaje imperativo.

2. Crear programas simples escritos en Java.

3. Comprender la misión y el funcionamiento de las distintas estructuras de control de flujo, realizar recorridos y búsquedas.

4. Definir funciones aplicando criterios simples de diseño descendente.

5. Crear clases simples, con sus constructores, atributos y funciones.

6. Entender los mecanismos de extensión y herencia.

7. Utilizar las clases del API de Java.

1. Fundamentos de programación

1.1. Declaración y uso de variables

La memoria de un ordenador permite almacenar y posteriormente recuperar gran cantidad de datos. Cuando se programa con un lenguaje de alto nivel como Java, se utilizan nombres para hacer referencia a los datos que hay en la memoria. En el argot del programador, a estos datos con nombre se les llama **variables**. Por definición, no puede haber dos variables con el mismo nombre.

Para garantizar una cierta coherencia en el uso de las variables, éstas, además de tener nombre, deben ser de un **tipo**. El tipo determina qué datos puede almacenar una variable (texto, números, etc.) y qué operaciones podemos realizar con ella. Empezaremos trabajando con los tipos siguientes:

- **int:** valor entero entre –2147483648 y 2147483647. No admite valores decimales. Por ejemplo, 2001 ó 175000000.
- **double:** valor real sin límite de rango y de precisión arbitraria. Admite valores decimales. Por ejemplo, –0,075 ó 3,141592654.
- **String:** texto de longitud arbitraria. Por ejemplo, "La casa de María" u "¡Hola, mundo!".
- **boolean:** cierto o falso.

Para poder trabajar con una variable, primero tenemos que declararla. Declarar una variable consiste en mencionar su nombre y su tipo. En Java se escribe primero su tipo, después su nombre y se termina la línea con un punto y coma. El punto y coma es importante, porque indica el final de la declaración. Por ejemplo:

```
int contador;
```

declara una variable llamada "contador" de tipo *int*. Como ya hemos visto anteriormente, en esta variable podremos almacenar un valor entre –2147483648 y 2147483647.

Las líneas siguientes:

```
boolean esClientePreferente;
double capital, interés;
```

declaran tres variables: "esClientePreferente", de tipo *boolean*, y "capital" e "interés", ambas de tipo *double*. En la primera podremos almacenar *cierto* o *falso*, lo que nos servirá, por ejemplo, para almacenar si se trata de un cliente preferente o no. En las dos últimas, almacenaremos el capital y el interés que estudiamos darle a ese mismo cliente, que puede ser diferente en virtud de que sea cliente preferente o no. Observad que podemos declarar dos o más variables en una misma línea separando sus nombres por medio de la coma.

La línea siguiente:

```
int valores[];
```

declara un vector de variables de tipo *int* llamado "valores". Un vector es un conjunto de variables del mismo tipo. Utilizaremos "valores[0]" para referirnos a la primera variable, "valores[1]" para referirnos a la segunda, y así sucesivamente.

En Java, se puede dar cualquier nombre a una variable. Sin embargo, se deben respetar unas pocas reglas. Los nombres de las variables:

- No pueden comenzar por un carácter numérico. Por ejemplo, "12" o "1Valor" no son nombres válidos.

- No pueden llamarse igual que los elementos del lenguaje. Por ejemplo, no podemos tener una variable que se llame "int" o "double".

- No pueden contener espacios.

Sin embargo, a diferencia de otros lenguajes, Java sí que permite que el nombre de una variable contenga caracteres acentuados.

Otro punto que se debe tener en cuenta es que Java distingue entre mayúsculas y minúsculas. Esto significa que las variables:

```
double capital, Capital;
```

son diferentes en todos los sentidos.

1.2. El operador de asignación

Una vez declarada una variable, ésta tiene un valor indefinido. Para empezar a trabajar con ella, se necesita asignarle un valor. La asignación de valores a

variables se lleva a cabo mediante el llamado operador de asignación: "=" (el signo de igual). Así pues, las líneas siguientes:

```
int contador;
contador = 0;
```

declaran la variable "contador" y le asignan el valor 0.

Como podemos observar, a la izquierda del operador de asignación se emplaza el nombre de la variable, y a la derecha, el valor que se desea asignarle. La asignación (como la declaración de variables) termina en punto y coma.

Veamos más ejemplos de asignaciones:

```
int suma;
suma = 3 + 2;
```

asigna a la variable "suma" el resultado de sumar 3 y 2 (cinco).

```
int contador;
contador = 0;
contador = contador + 1;
```

asigna, en primer lugar, el valor 0 a la variable "contador", y luego la incrementa en una unidad (1). Es importante destacar que cuando asignamos un valor a una variable sobrescribimos su valor anterior. Así pues, al finalizar el código, la variable "contador" valdrá 1.

```
double interés, capital, tasaInterés, tiempo;
capital = 60000;
tasaInterés = 0.045;
tiempo = 6;
interés = capital * tasaInterés * tiempo;
```

asigna a la variable "interés" los intereses generados por el préstamo de un capital de 60.000 euros a una tasa del 4,5% (expresado en tanto por uno) a seis años. Según la fórmula del interés simple, los intereses serán igual al producto de los tres factores. Como puede apreciarse, el producto se representa mediante el carácter asterisco (*). Al final de la ejecución, "interés" valdrá 16.200.

```
String nombre, frase;
nombre = "María";
frase = "La casa de " + nombre;
```

asigna a la variable “frase” el resultado de concatenar el texto “La casa de ” y la variable “nombre”. Al finalizar la ejecución, “frase” será igual a “La casa de María”. Debemos notar que el texto siempre se escribe encerrado entre comillas dobles (“”).

```
boolean esPositivo;
int número;
número = -5;
es_positivo = número > 0;
```

almacena en la variable “esPositivo” el resultado de evaluar si la variable “número” es mayor que 0. Al finalizar el código, “esPositivo” valdrá *false* (falso).

Si bien la flexibilidad que ofrece el operador de asignación es notable, para que una asignación sea válida debe haber una cierta compatibilidad entre el tipo de la variable y el tipo del valor que se le quiere asignar. Por ejemplo, la asignación siguiente no es válida:

```
int peso;
peso = 58.5;
```

pues intenta asignar un valor decimal a una variable entera.

Tampoco es válida la asignación:

```
String resultado;
resultado = 25;
```

pues intenta asignar un valor entero a una variable de tipo texto.

1.3. Expresiones

Como hemos visto en los ejemplos anteriores, a una variable no sólo se le puede asignar un valor literal o el valor de otra variable. Además, se le puede asignar el valor resultante de una operación. Cuando nos encontramos con una combinación de valores literales, variables y operadores, estamos ante lo que se llama una

expresión. Las expresiones, que siempre generan un valor, están sometidas a unas reglas de evaluación que debemos conocer. Si tenemos, por ejemplo:

```
int cálculo;
cálculo = 2 + 3 * 5;
```

la expresión “2 + 3 * 5” se puede interpretar de dos formas: sumar 2 y 3, y el resultado (5) multiplicarlo por 5 (25), o bien multiplicar 3 por 5, y al resultado (15) sumarle 2 (17). Como podemos ver, de la forma de interpretar la expresión depende el resultado.

El lenguaje Java, como muchos otros lenguajes, define unas prioridades o precedencias entre operadores que dictan el orden en el que se realizarán las operaciones. La tabla “Precedencia de los operadores” muestra los operadores más comunes y su precedencia. Cuanta mayor precedencia, antes se evaluará la operación.

Si aplicamos las reglas de precedencia de la tabla al ejemplo anterior, tenemos que la multiplicación precederá a la suma. En consecuencia, el lenguaje resolverá la expresión multiplicando primero 3 por 5, y sumándole 2 al resultado. Así pues, se asignará el valor 17 a la variable “cálculo”.

<table>
<tr><th>Precedencia</th><th colspan="4">Operadores</th><th>Descripción</th></tr>
<tr><td>Más precedencia</td><td>–expr.</td><td>+expr.</td><td colspan="2">!</td><td>Signo positivo, signo negativo y no.</td></tr>
<tr><td></td><td>*</td><td>/</td><td colspan="2">%</td><td>Multiplicación, y cociente y residuo de una división.</td></tr>
<tr><td></td><td colspan="2">+</td><td colspan="2">–</td><td>Suma/concatenación* y resta.</td></tr>
<tr><td></td><td><</td><td>></td><td><=</td><td>>=</td><td>“Más pequeño que”, “más grande que”, “más pequeño o igual que” y “más grande o igual que”.</td></tr>
<tr><td></td><td colspan="2">==</td><td colspan="2">!=</td><td>“Igual que” y “diferente que”.</td></tr>
<tr><td></td><td colspan="4">&&</td><td>Y</td></tr>
<tr><td>Menos precedencia</td><td colspan="4">||</td><td>O</td></tr>
</table>

Tabla 1. Precedencia de los operadores

*** El significado del operador “+” cambia dependiendo de los operandos. Aplicado a números realiza una suma; aplicado a texto, una concatenación.**

Como hemos visto, la precedencia entre operadores permite al lenguaje determinar, de forma unívoca, el valor de una expresión. Sin embargo, este mecanismo no es suficiente en el caso de tener en una expresión dos o más operadores con la misma precedencia. Cuando esto sucede, el lenguaje resuelve la expresión evaluando dichos operadores de derecha a izquierda. Es decir, se operarán en el mismo sentido en el que se leen. Por ejemplo, el código:

```
int cálculo;
cálculo = 25 % 7 * 3;
```

calculará primero el residuo resultante de dividir 25 entre 7, y después multiplicará dicho residuo por 3. El resultado será 12.

Supongamos ahora que deseamos sumar 2 y 3, y multiplicar el resultado por 5. Ya hemos visto que el código:

```
int cálculo;
cálculo = 2 + 3 * 5;
```

no responde a nuestras necesidades, pues, por precedencia, se evaluará antes la multiplicación que la suma. Para alterar el orden de evaluación de la expresión, encerraremos la suma entre paréntesis:

```
int cálculo;
cálculo = (2 + 3) * 5;
```

Cuando encerramos parte de una expresión entre paréntesis, esta parte se considera una unidad indivisible y se evalúa independientemente antes de ser operada. Así, volviendo al ejemplo, la expresión se resolverá multiplicando el resultado de evaluar la expresión “2 + 3” por 5. El resultado será 25.

Veamos más ejemplos de expresiones. Las líneas siguientes:

```
double precioConDescuento, precio, descuento;
precio = 15.69; // en euros
descuento = 7; // tanto por ciento
precioConDescuento = precio – precio * descuento / 100;
```

calculan el precio rebajado de un producto. En primer lugar, por tener más precedencia, se hace la multiplicación (precio * descuento), después se divide el resultado por cien, y finalmente, se resta el resultado obtenido del valor de la variable “precio”. El resultado será 14,5917.

Las líneas siguientes:

```
int número;
boolean estáEntreCeroYDiez;
número = –5;
estáEntreCeroYDiez = número > 0 && número < 10;
```

determinan si un número dado está entre 0 y 10. Para que esto se cumpla, el número debe ser, a la vez, mayor que 0 y menor que 10. En primer lugar, se comprobará si el número es mayor que 0 (número > 0), y después, si es menor que 10 (número < 10), para, finalmente, mediante el operador && (y), ver si ambas condiciones se cumplen. El resultado será igual a *false* (falso).

Aunque el conocimiento y el buen uso de las reglas de precedencia delatan el buen hacer de un programador, a veces, un paréntesis en el lugar adecuado o el uso de expresiones más simples ayudan a despejar dudas y hacer el código mucho más legible. En consecuencia, debemos llegar a un compromiso entre el uso de la precedencia y los paréntesis, en pro de la legibilidad del código.

1.4. Estructura básica de un programa

El lenguaje Java, como la mayoría de lenguajes de programación, requiere una estructura mínima que acoja el programa y sus componentes. En el caso concreto de Java, a esta estructura mínima se le llama **clase**. Aunque las clases se estudiarán en detalle en el capítulo 2, es necesario conocer esta estructura para empezar a escribir programas funcionales. A continuación, se muestra la estructura mínima de una clase ejecutable:

```
public class Programa {
  public static void main(String [] args) {
    // aquí va el código
  }
}
```

Sin querer entrar en demasiados detalles, el código define una clase (*class*) llamada "Programa" con una función "main" que será el que se ejecutará al iniciar el programa. El nombre de la clase ha sido escogido arbitrariamente (es decir, puede cambiarse); no así el nombre de la función, que siempre deberá llamarse "main". Aunque las funciones se estudiarán en detalle en el apartado 1.7, es necesario adelantar que son simples agrupaciones (funcionales) de código. Este programa deberá guardarse en un fichero que se llame como la clase más el sufijo *.java*; en este caso, "Programa.java". El código del programa deberá ir entre las llaves de la función "main".

El programa siguiente:

```
public class Programa {
  public static void main(String [] args) {
    double distancia, velocidad, tiempo;
    velocidad = 120; // en km/h
    tiempo = 2.5; // en horas
    distancia = velocidad * tiempo; // 300 km
    System.out.println(distancia); // escribe 300
  }
}
```

calcula la distancia recorrida por un bólido a partir de su velocidad y el tiempo que ha estado circulando. El texto precedido por doble barra (//) son comentarios. Los comentarios son simples anotaciones que el programador hace en el código del programa para facilitar su comprensión. Los comentarios no se ejecutan. La última línea usa la función "System.out.println" para escribir el resultado en la pantalla.

Como puede apreciarse, todas las variables se han declarado al inicio del código. A diferencia de otros lenguajes, en Java las variables pueden declararse en cualquier parte del código, siempre y cuando la declaración preceda a su uso. Sin embargo, es una práctica habitual declararlas al principio del código para mejorar su legibilidad.

1.5. Estructuras de control de flujo

Supongamos que se desea escribir un programa que calcule y muestre por pantalla el resultado de la suma de los números pares entre 0 y 10. El programa se podría escribir así:

```
public class SumaPares {
  public static void main(String [] args) {
    int suma;
    suma = 2 + 4 + 6 + 8;
    System.out.println(suma); // escribe 20
  }
}
```

El programa, efectivamente, escribirá "20". Sin embargo, si se deseara conocer la suma de los números pares entre 0 y 10.000, escribirlos todos sería inviable o, cuando menos, demasiado costoso.

La solución a este problema nos la dan las estructuras de control de flujo del programa, que nos permiten repetir instrucciones o ejecutar unas u otras dependiendo de una condición.

Para sumar los números pares entre 0 y 10.000, en primer lugar, necesitamos generar de alguna manera todos esos números pares. Una forma fácil de hacerlo es partiendo del primer par, en este caso el 2, e irle sumando 2 una y otra vez:

```
númeroPar = 2;
númeroPar = númeroPar + 2; // númeroPar = 4
númeroPar = númeroPar + 2; // númeroPar = 6
númeroPar = númeroPar + 2; // númeroPar = 8
```

hasta llegar al último par:

```
númeroPar = númeroPar + 2; // númeroPar = 9998
```

en total, 4.998 líneas de código, mientras “númeroPar” sea más pequeño que 10.000. Afortunadamente, esto se puede reducir a tres líneas usando una estructura *while* (mientras):

```
númeroPar = 2;
while (númeroPar < 10000) { // mientras < 10000
  númeroPar = númeroPar + 2; // siguiente número par
}
```

La estructura *while* repite una serie de instrucciones mientras se cumpla una condición dada. Dicha condición se cierra entre paréntesis a la derecha del *while*, mientras que las instrucciones que se han de repetir se cierran entre llaves ({}). En el ejemplo, se repetirá la suma mientras “númeroPar” sea más pequeño que 10.000.

Volviendo a nuestro empeño por conocer el valor de la suma de todos estos números, una vez localizados tan sólo nos queda irlos sumando. Para hacer esto, declararemos la variable “suma”, que irá acumulando el resultado. Sólo debemos tener la precaución de ponerla a cero antes de empezar. El programa queda así:

```
public class SumaPares2 {
  public static void main(String[] args) {
    int suma, númeroPar;

    suma = 0;
    númeroPar = 2; // el primer número par
    while (númeroPar < 10000) { // mientras < 10000
      suma += númeroPar;
      númeroPar += 2; // siguiente número par
    }

    System.out.println(suma);
  }
}
```

Al final del programa, se mostrará el resultado por pantalla (24.995.000). Hemos de destacar el uso del operador “+=”, que asigna a una variable el valor de ella misma más el valor especificado. Es decir:

```
númeroPar += 2;
```

equivale a:

```
númeroPar = númeroPar + 2;
```

Supongamos ahora que queremos crear un programa que sume los múltiplos de un número dado en un intervalo dado. Por ejemplo, los múltiplos de 122 entre 20.000 y 40.000. El programa que suma los múltiplos de dos tiene una limitación: necesita conocer el primer múltiplo del intervalo para poder generar los demás.

Reconocer el primer número par de un intervalo es muy fácil, pero las cosas se complican bastante cuando es cuestión de encontrar el primer múltiplo de 122.

El programa "SumaPares2" construía cada número par sumando dos al inmediatamente anterior. Una vez generado, lo acumulaba, y así hasta haber agotado todos los números pares del intervalo. Sin embargo, ésta no era la única solución: se podría haber optado por recorrer todos los números del intervalo, determinando cuáles son pares y cuáles no. Si el número es par se acumula; si no, no. Esta dicotomía (si se cumple condición, hacer A; si no, hacer B) se puede modelar mediante la estructura *if/else* (si/si no).

Una forma de determinar si un número dado es par o no, es dividiéndolo por dos y analizando el residuo. Si el residuo es cero, el número es par. Teniendo en cuenta que en Java el residuo de una división se obtiene mediante el operador "%" (por cien), esta condición se podría escribir así:

```
if (número % 2 == 0) {
  // código que hay que ejecutar si es par
}
else {
  // código que hay que ejecutar si no es par
}
```

De modo similar a la estructura *while*, la estructura *if* ejecuta una serie de instrucciones si se cumple una determinada condición. Dicha condición se cierra entre paréntesis a la derecha del *if*, mientras que las instrucciones que se han de ejecutar se cierran entre llaves ({}). Opcionalmente, la estructura *if* acepta una extensión *else*, que especifica una serie de instrucciones que se han de ejecutar en caso de que la condición no se cumpla.

Volviendo al problema, si el número es par, debemos acumularlo, mientras que si no lo es, simplemente lo ignoramos.Utilizando la estructura *if*, el código quedaría así:

```
public class SumaPares3 {
  public static void main(String [] args) {
  int número, suma;
  número = 1; // primer número del intervalo
  suma = 0;
  while (número <= 9999) { // último número del intervalo
    if (número % 2 == 0) { // ¿es par?
      suma += número; // sí, entonces lo suma
    }
    número += 1; // siguiente número del intervalo
  }

    System.out.println(suma);
}
```

Al igual que “SumaPares2”, “SumaPares3” mostrará 24.995.000 por pantalla. Sin embargo, modificar esta última versión para que opere en otro rango o para que sume los múltiplos de un número diferente de 2, es inmediato. Siguiendo con el ejemplo, si se desea sumar los múltiplos de 122 entre 20.000 y 40.000, basta con cambiar sólo tres líneas:

```
public class SumaMultiples {
  public static void main(String [] args) {
  int número, suma;

  número = 20001; // primer número del intervalo
  suma = 0;
  while (número <= 39999) { // último número
    if (número % 122 == 0) { // ¿es múltiplo de 122?
      suma += número; // sí, entonces lo suma
    }
    número += 1; // siguiente número del intervalo
  }

  System.out.println(suma);
}
```

1.6. Esquemas de recorrido y de búsqueda

Hemos visto que cuando se trabaja con estructuras del tipo *while*, se repiten una serie de instrucciones mientras se cumpla una determinada condición. Generalmente, se recurre a las repeticiones con uno de estos dos objetivos:

- Recorrer todos los elementos de un conjunto.
- Buscar un elemento de un conjunto.

Este hecho nos lleva a hablar de esquemas de recorrido y de búsqueda. En el ejemplo "SumaPares3", se analizan los números entre 1 y 9.999. Se comprueba cada número para ver si es par o no y, si lo es, se acumula. No se sale del bucle hasta que no se han analizado todos y cada uno de los números del rango. Estamos pues ante un esquema de recorrido.

Los esquemas de recorrido tienen siempre la misma estructura:

```
while (haya_elementos) {
  hacer_acción(elemento_actual);
  siguiente_elemento();
}
```

Por otro lado, podemos tener repeticiones que tengan como objetivo buscar un determinado elemento dentro de un conjunto. Sin apartarnos de los ejemplos de carácter numérico, supongamos que se desea encontrar el primer múltiplo de 13 entre 789 y 800. Está claro que se deben comprobar los números del rango hasta que se encuentre el primer múltiplo de 13. El código siguiente podría resolver el problema:

```
int número;

número = 789;
while (número % 13 != 0) {
  número++;
}
```

Básicamente, empezando por 789, se va incrementando la variable "número" ("número++" equivale a "número = número + 1") mientras "número" no sea múltiplo de 13. Sin embargo, esta aproximación comete un error: supone que encontrará algún múltiplo dentro del rango. Si este múltiplo no existiera, el bucle *while* continuaría analizando números más allá del rango del problema. Por esto, además de comprobar que no se ha encontrado el número que se buscaba, también se debe comprobar que no se esté saliendo del rango de análisis. El código siguiente tiene en cuenta ambas cosas:

```
int número;

número = 789;
while (número <= 800 && número % 13 != 0) {
  número++;
}
```

Como se puede observar, en primer lugar, se comprueba que "número" no sea mayor que 800 (es decir, que sea más pequeño o igual a 800) y, en segundo lugar, que no sea múltiplo de 13. Al haber dos condiciones que gobiernan el bucle (número <= 800 y número % 13 != 0), se puede salir de él por dos motivos:

- por haber agotado todos los números del rango ("número" ha alcanzado el valor 801) o

- por haber encontrado un múltiplo de 13.

Por esta razón, debemos comprobar, al finalizar el bucle, si se ha salido de él por un motivo u otro, pues de ello va a depender el comportamiento del resto del programa:

```
int número;

número = 789;
while (número <= 800 && número % 13 != 0) {
  número++;
}

if (número % 13 != 0) { // se ha encontrado un múltiplo
  System.out.println(número);
}
else { // no se ha encontrado un múltiplo
  System.out.println("No se ha encontrado.");
}
```

Los esquemas de búsqueda presentan la estructura siguiente:

```
while (haya_elementos && ! elemento_encontrado) {
  siguiente_elemento();
}
```

Es importante observar que la condición "elemento_encontrado" siempre irá negada. Esto siempre será así porque las instrucciones del bucle deben repetirse mientras no se haya encontrado el elemento que se busca.

1.7. Definición y uso de funciones

En nuestro afán por encontrar números de todo tipo, supongamos ahora que deseamos encontrar el primer número primo de un intervalo dado. Hay que recordar que un número primo es aquel que sólo se divide de forma entera por él mismo y por 1. El 13 o el 19, por ejemplo, son números primos.

El siguiente esquema de búsqueda podría resolver el problema:

```
encontrado = es_primo el primer_número_del_intervalo;
while (!encontrado && hay_mas_numeros_en_el_intervalo) {
  encontrado = es_primo el siguiente_número;
}

if encontrado {
  // número encontrado
}
else {
  // número no encontrado
}
```

Probar que un número es primo no es una tarea fácil. De hecho, dado un número, llamémosle *N*, la única forma de comprobar que es primo es dividiéndolo entre todos los números primos entre 2 y *N* – 1, ambos incluidos*. Para que sea primo, los residuos de las divisiones deben ser diferentes de 0. Si hay alguna división cuyo residuo sea 0, el número no es primo.

*** Realmente, hay otras formas mucho más optimas de comprobar la primalidad de un número. Véase http://es.wikipedia.org/wiki/Test_de_primalidad.**

Para simplificar el problema, probaremos la primalidad de un número dividiéndolo entre todos los números menores que él, sean primos o no.

Vamos a generar, en primer lugar, un código que compruebe si un número es primo. Concretamente, intentaremos demostrar que no lo es buscando un divisor entero del mismo. Si fallamos en nuestro intento, el número será primo. Como ya hemos visto, se trata de ir dividiendo el número en cuestión entre todos los números menores que él hasta obtener una división cuyo residuo sea 0. Es decir, debemos implementar un esquema de búsqueda del menor divisor entero:

```
int número, divisor;
boolean esPrimo;

número = 19; // número que queremos comprobar

divisor = 2;
while (número % divisor != 0 && divisor < número) {
```

```
  divisor ++;
}
esPrimo = divisor == número; // resultado: true o false
```

Como en todos los recorridos de búsqueda, hay dos condiciones que gobiernan el *while*: la que comprueba que "divisor" no sea divisor entero de "número" (número % divisor != 0) y la que comprueba que no se haya salido de rango (divisor < número). Consecuentemente, al terminar la ejecución del bucle, "número" será primo si la condición que ha "fallado" ha sido la segunda. En este caso, el número será primo si "divisor" es igual a "número".

Ahora que ya disponemos de un código capaz de determinar si un número dado es primo, vamos a construir el programa que busca el primer número primo de un intervalo. Basándonos en el pseudocódigo propuesto anteriormente, el código siguiente busca el primer número primo entre 10.000 y 10.013:

```
public class BuscarPrimo {
 public static void main(String[] args) {
  int inicio, fin; // inicio y fin del rango
  int número, divisor;
  boolean encontrado;

  // inicio y fin del intervalo de búsqueda
  inicio = 10000;
  fin = 10013;

  System.out.println("Buscando el primer número " +
     " primo entre " + inicio + " y " + fin + "...");
  // comprueba si el primer número del intervalo es
  // primo
  número = inicio; // número que hay que comprobar
  divisor = 2;
  while (número % divisor != 0 && divisor < número) {
   divisor ++;
  }
  encontrado = (divisor == número); // resultado

  // mientras no se haya encontrado un número primo y
  // haya más números en el intervalo
  while (! encontrado && número < fin) {
   // comprueba el siguiente número
```

```
            número ++;
            divisor = 2;
            while (número % divisor != 0 && divisor < número) {
              divisor ++;
            }
            encontrado = (divisor == número); // resultado
          }
          if (encontrado) {
            // si ha encontrado un número primo lo muestra por
            // pantalla
            System.out.println(número);
          }
          else {
            System.out.println("No hay ningún número primo " +
                "en el intervalo");
          }
        }
      }
```

Básicamente, el código es un esquema de búsqueda del primer número primo, combinado con el código que comprueba si un número es primo visto anteriormente. Como se puede apreciar, éste último, que hemos enmarcado para distinguirlo con facilidad, se repite dos veces.

Repetir bloques de código dos o más veces dentro de un programa tiene serios inconvenientes:

- **Reduce la legibilidad.** Como se puede apreciar en el programa, cuesta discernir entre el código que pertenece a la búsqueda del número primo y el que comprueba si un número lo es. Las variables se mezclan y las instrucciones también.

- **Dificulta la mantenibilidad.** Si se detecta un error en la función que comprueba si un número es primo o se desea mejorarlo, las modificaciones deben propagarse por todos y cada uno de los bloques repetidos. Cuantas más repeticiones, más difícil será introducir cambios.

- **Es propenso a errores.** Ello es consecuencia de la baja legibilidad y mantenibilidad. Una corrección o mejora que no se ha propagado correctamente por todos los fragmentos repetidos puede generar nuevos errores. La corrección de éstos, a su vez, puede generar otros nuevos, y así indefinidamente.

Al final, la repetición de bloques de código conduce a códigos inestables con un coste de mantenimiento más elevado en cada revisión. Afortunadamente, todos estos inconvenientes se pueden salvar mediante el uso de funciones.

Si nos fijamos en el pseudocódigo que hemos concebido inicialmente, vemos que, de forma natural, hemos detectado la funcionalidad "comprobar si un número es primo", que hemos expresado mediande la fórmula "es_primo". Más adelante, cuando hemos implementado dicha funcionalidad, ha aparecido una variable que podríamos considerar la entrada del subprograma: "número", y otra que se podría considerar su resultado: "esPrimo". En la implementación final, el programa principal ha hecho uso de ambas variables para comunicarse con el subprograma, pasando el número a comprobar a través de la variable "número" y obteniendo el resultado mediante la variable "encontrado" (que es el nuevo nombre que, por razones de legibilidad, hemos dado a la variable "esPrimo").

De todo esto, se derivan dos conclusiones importantes:

- La lógica del subprograma es completamente independiente de la del programa principal. Es decir, al programa principal poco le importa cómo se comprueba si un número es primo, ni que variables se utilizan para hacer dicha comprobación.

- Programa y subpgorama se comunican mediante una interfaz basada en variables bien definida.

Llegados a este punto, podemos declarar lo que se llama una *función*. Una función no es más que una agrupación de código que, a partir de uno o más valores de entrada, o, valga la redundancia, en función de ellos, genera un resultado.

Veamos pues la función "esPrimo":

```
static boolean esPrimo(int número) {
 int divisor;

 divisor = 2;
 while (número % divisor != 0 && divisor < número) {
  divisor ++;
 }

 return divisor == número; // resultado: true o false
}
```

Al margen del uso del modificador *static*, cuyo significado se estudiará en el capítulo 2, una función se declara escribiendo el tipo de resultado que devuelve (en este caso *boolean*), su nombre ("esPrimo") y, encerrados entre paréntesis, sus parámetros de entrada separados por comas ("número" de tipo *int*). Como se aprecia en el código, los parámetros de entrada se definen de la misma forma que las variables: en primer lugar se escribe su tipo y, luego,

su nombre. El código de la función que sigue a este bloque declarativo va encerrado entre llaves.

Podemos observar que al final de la función aparece una sentencia nueva: *return* (devolver). Cuando se alcanza un *return*, termina la ejecución de la función, y se establece como resultado de la misma el valor de la expresión que sigue al *return*. La sentencia *return* siempre termina en punto y coma.

Una vez definida la función, podemos llamarla desde cualquier parte del código. Por ejemplo, la línea siguiente:

```
primo = esPrimo(19);
```

determina si el número 19 es primo. Dicho de otra manera, asigna a la variable "primo" (de tipo *boolean*) el resultado de la función "esPrimo" para el valor 19. La llamada a una función también termina en punto y coma. Es importante observar que, al ejecutarse la función, el parámetro "número" valdrá 19. De la misma forma, al terminarla, se asignará a la variable "primo" el valor de la expresión que hay a la derecha del *return*:

```
static boolean esPrimo(int número) {
  int divisor;

  divisor = 2;
  while (número % divisor != 0 && divisor < número) {
    divisor ++;
  }

  return divisor == número; // resultado: true o false
}
```

❷ ❶

```
primo = esPrimo(19);
```

También es importante destacar que cuando se llama a una función pasa a ejecutarse el código de ésta, y no se vuelve al código que la ha llamado hasta que no se alcanza un *return*.

Si rescribimos el programa "BuscarPrimo" usando funciones, nos queda así:

```
public class BuscarPrimo2 {
  static boolean esPrimo(int número) {
```

```
    int divisor;

    divisor = 2;
    while (número % divisor != 0 && divisor < número) {
      divisor ++;
    }

    return divisor == número; // resultado: true o false
  }

  public static void main(String[] args) {
    int inicio, fin; // inicio y fin del rango
    int número;
    boolean encontrado;

    // inicio y fin del intervalo de búsqueda
    inicio = 10000;
    fin = 10013;

    System.out.println("Buscando el primer número " +
        " primo entre " + inicio + " y " + fin + "...");

    // comprueba si el primer número del intervalo es
    // primo
    número = inicio; // número que hay que comprobar
    encontrado = esPrimo(número); // resultado

    // mientras no se haya encontrado un número primo y
    // haya más números en el intervalo
    while (! es_primo && número <= fin) {
      // comprueba el siguiente número
      número ++;
      encontrado = esPrimo(número); // resultado
    }

    if (encontrado) {
      // si ha encontrado un número primo lo muestra por
      // pantalla
      System.out.println(número");
    }
    else {
      System.out.println("No hay ningún número primo " +
         "en el intervalo");
    }
  }
}
```

Incluso podemos simplificar el recorrido de búsqueda aún más:

```
número = inicio;
while (número <= fin && ! esPrimo(número)) {
  número ++;
}
```

Cuando se trabaja con funciones, deben tenerse en cuenta los aspectos siguientes:

- Los parámetros de una función actúan a todos los efectos como variables. Podemos leerlos y operar con ellos como si fueran variables.

- Los parámetros de una función son una copia de los parámetros de entrada. Si pasamos una variable como parámetro a una función, y dentro del código de la función se modifica el parámetro, el valor de la variable no se verá alterado.

- Al espacio entre llaves que encierra el código de la función se le llama "ámbito de la función". Las variables declaradas dentro de la función sólo pueden verse dentro del ámbito de la función. Es decir, no se puede acceder a ellas desde fuera de la función. Lo mismo es aplicable a los parámetros.

- Una función se caracteriza a partir de su nombre y el tipo de los parámetros de entrada. Si cambia alguno de ellos, estamos ante funciones diferentes. Esto significa que puede haber funciones con el mismo nombre pero con parámetros distintos. A esta característica, que no todos los lenguajes soportan, se le llama "sobrecarga de funciones". A partir del número y el tipo de los parámetros de entrada en una llamada, se averiguará la función a la que se llama.

- Una función puede que no devuelva ningún resultado. En este caso, el tipo de la función será `void` (vacío), y estaremos ante lo que se llama un procedimiento. Un ejemplo de procedimiento és la función `main`, que es la primera función que se ejecuta al iniciar un programa.

2. Programación orientada a objetos

A lo largo de la historia de la informática, han ido apareciendo diferentes paradigmas de programación. En primer lugar, apareció la programación secuencial, que consistía en secuencias de sentencias que se ejecutaban una tras otra. El lenguaje ensamblador o el lenguaje COBOL son lenguajes secuenciales. Entonces no existía el concepto de función, que apareció más adelante en los lenguajes procedimentales, como el BASIC o el C. La evolución no terminó aquí, y continuó hasta el paradigma más extendido en la actualidad: la programación orientada a objetos. Smalltalk, C++ o Java pertenecen a esta nueva generación.

Cada nuevo paradigma ha extendido el anterior, de manera que podemos encontrar las características de un lenguaje secuencial en uno procedimental, y las de uno procedimental en uno orientado a objetos. De hecho, hasta ahora sólo hemos visto la vertiente procedimental del lenguaje Java.

En este capítulo, estudiaremos qué es una clase y qué es un objeto, cómo se definen y cómo se utilizan. Además, conoceremos los aspectos más relevantes de la programación orientada a objetos y las principales diferencias respecto a la programación procedimental.

2.1. Clases y objetos

En la programación orientada a objetos, aparecen por primera vez los conceptos de clase y objeto. Una clase es como una especie de patrón conceptual, mientras que un objeto es la materialización de dicho patrón. Imaginemos la clase "motocicleta". Todos estamos de acuerdo en que todas las motocicletas tienen características comunes que nos permiten distinguirlas como tales. Una motocicleta, entre otras cosas, tiene dos ruedas, carece de techo y tiene un manillar y un motor de una determinada cilindrada. Además, será de alguna marca y tendrá algún nombre de modelo. A este esquema mental, a este patrón, lo llamaremos "clase". Sin embargo, motocicletas hay muchas: la de mi hermano, la del vecino, la del concesionario de enfrente..., todas de diferentes marcas, cilindradas y colores. A esta materialización del patrón le daremos el nombre de "objeto". Clase sólo hay una, pero objetos puede haber muchos.

Una vez aclarada la diferencia entre una clase y un objeto, pasemos a ver cómo declarar una clase. Para nuestro propósito, tomaremos como ejemplo la clase "Producto" para representar cualquier producto que puede venderse en un colmado. Nuestro objetivo, al final, será imprimir un tíquet de compra.

Aunque los productos que podemos encontrar en un colmado tienen características muy dispares, sí que comparten unas pocas que nos permitirán alcan-

zar nuestro objetivo: un código de producto (que corresponderá al código de barras), una descripción y un precio (por ejemplo, un frasco de café soluble, "café de la UOC", con el código 8000534569044 y un precio de 3,50 euros).

Estas características, que llamaremos atributos, se pueden representar como variables. Las líneas siguientes declaran la clase "Producto" y sus atributos:

```
class Producto {
  int código;
  String descripción;
  double precio;
}
```

Como se puede observar, el nombre de la clase va precedido por la palabra reservada *class* (clase). Los atributos se declaran del mismo modo que las variables, aunque encerrados entre llaves.

Cabe recordar que una clase es siempre una simplificación de la realidad. Por esta razón, nunca declararemos atributos para todas y cada una de las características del ente que queramos representar; sólo crearemos aquellos que necesitemos para resolver el problema que se nos plantea.

Por otro lado, podemos tener una o más funciones que lean y escriban esos atributos, por ejemplo, una función que se encargue de leer y otra que se encargue de cambiar (escribir) el precio del producto. Estas funciones, que están estrechamente ligadas a los atributos y que no tienen razón de ser sin ellos, se declaran en el ámbito de la clase.

El código siguiente:

```
public class Producto {
  int código;
  String descripción;
  double precio;

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }
}
```

declara la clase “Producto” con dos funciones miembro: “fijarPrecio” y “obtenerPrecio”. Es importante observar que ambas funciones tienen visibilidad sobre los atributos de la clase, es decir, pueden acceder a ellos de la misma forma que acceden a sus propios parámetros y variables.

Observad que las funciones que hemos definido carecen del modificador *static* (estático) que hemos visto por primera vez en el apartado 1.7 (“Definición y uso de funciones”). Sin querer entrar en demasiados detalles (cuya discusión va más allá de los objetivos de estos materiales), las funciones que quieran acceder a los atributos de una clase no pueden ser estáticas. En consecuencia, deben prescindir del modificador *static*.

2.2. Instanciación de objetos

Como ya hemos comentado, una clase es sólo un patrón de objetos. Los atributos de una clase no existen en la memoria del ordenador hasta que no se materializan en un objeto. A este proceso de materialización se le llama instanciación.

Un objeto se instancia mediante el operador *new* (nuevo).

El código siguiente:

```
Producto sal;
sal = new Producto();
```

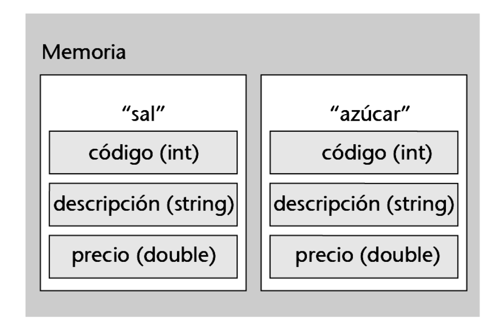
instancia el objeto “sal” de la clase “Producto”. Como se puede observar, “sal” debe declararse como una variable del tipo “Producto”. Los paréntesis después del nombre de la clase son obligatorios.

A partir de la instanciación, ya tenemos en memoria un objeto con todos sus atributos. Cada nueva instanciación crea un objeto nuevo, independiente de los demás. Si tenemos, por ejemplo, dos objetos, “sal” y “azúcar”:

```
Producto sal, azúcar;

sal = new Producto();
azúcar = new Producto();
```

en la memoria tendremos dos pares de atributos “código”, “descripción” y “precio”, uno por cada objeto:



Para acceder a estos atributos, utilizaremos el operador "." (punto).

Las líneas siguientes:

```
Producto sal, azúcar;

sal = new Producto();
azúcar = new Producto();

// fija el precio del paquete de sal
sal.precio = 0.60;

// fija el precio del paquete de azúcar
azúcar.precio = 0.81;
```

inicializan el atributo "precio" de los objetos "sal" y "azúcar" a 0,60 y 0,81 respectivamente. Observad la independencia entre ambos objetos. El operador "." nos permite especificar de forma unívoca a qué atributo de qué objeto hacemos referencia.

De la misma forma (usando el operador ".") accederemos a las funciones miembro de la clase. El código siguiente hace exactamente lo mismo que el anterior, pero usando la función "fijarPrecio" que anteriormente habíamos definido:

```
Producto sal, azúcar;

sal = new Producto();
azúcar = new Producto();

// fija el precio del paquete de sal
sal.fijarPrecio(0.60);

// fija el precio del paquete de azúcar
azúcar.fijarPrecio(0.81);
```

Agrupándolo todo, el código quedaría así:

```
public class Producto {
  int código;
  String descripción;
  double precio;

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }

  public static void main(String[] args) {
    Producto sal, azúcar;

    sal = new Producto();
    azúcar = new Producto();

    // fija el precio del paquete de sal
    sal.fijarPrecio(0.60);

    // fija el precio del paquete de azúcar
    azúcar.fijarPrecio(0.81);
  }
}
```

Observad que las funciones miembro de una clase, como “fijarPrecio” u “obtenerPrecio”, acceden directamente a los atributos de un objeto sin necesidad de especificar a qué objeto pertenecen. Esto es así porque las funciones miembro siempre se ejecutan en el contexto de un objeto. Es decir, cuando nos encontramos con la llamada:

```
sal.fijarPrecio(0.60);
```

ejecutamos la función para el objeto “sal”. Entonces, dentro de la función, cada vez que se hace referencia a un atributo, a éste, implícitamente, se le supone de la clase “sal”.

2.3. El objeto *this*

Como ya hemos explicado, cuando una función miembro accede a un atributo lo hace en el contexto de un objeto. Por eso no es necesario que especifique a qué objeto hace referencia. Sin embargo, hay casos en los que puede ser interesante disponer de dicho objeto, por ejemplo para pasarlo por parámetro a otra función o para distinguirlo de una variable o un parámetro homónimo.

Las funciones miembro pueden acceder al objeto actual mediante el objeto predefinido *this* (éste). A continuación, se muestra una implementación alternativa (y equivalente) de la función “fijarPrecio”:

```
void fijarPrecio(double precio) {
  this.precio = precio;
}
```

En este caso, se utiliza *this* para distinguir entre el atributo de la clase y el parámetro de la función. El “this.precio” hace referencia al atributo “precio” del objeto actual (*this*), mientras que “precio”, sin el *this*, hace referencia al parámetro de la función. En este caso, mediante *this* explicitamos a qué objeto hacemos referencia. Entonces, si tenemos la llamada:

```
sal.fijarPrecio(0.60);
```

en el contexto de la función “fijarPrecio”, *this* será el mismo objeto que “sal”.

2.4. El constructor

Cuando se instancia un objeto, puede ser necesario inicializar sus atributos. Volviendo al ejemplo anterior, establecíamos el precio del producto después de haberlo instanciado, ya fuese accediendo directamente al atributo “precio” o mediante la función “fijarPrecio”. Sin embargo, las clases nos ofrecen un mecanismo mucho más elegante de inicializar un objeto: el constructor.

El constructor es una función como cualquier otra, salvo por un par de particularidades: se llama como la clase y no tiene tipo de retorno.

Las líneas siguientes muestran un posible constructor de la clase “Producto”:

```
Producto(int código, String descripción, double precio) {
  this.código = código;
  this.descripción = descripción;
  this.precio = precio;
}
```

Observad que el constructor espera tres parámetros: el código, la descripción y el precio del producto. En este caso, la tarea que lleva a cabo el constructor es relativamente sencilla: tan sólo copiar el código, la descripción y el precio proporcionados a los atributos homónimos.

El ejemplo siguiente instancia otra vez los productos “sal” y “azúcar”. Sin embargo, esta vez se hace uso del constructor que acabamos de definir:

```
Producto sal, azúcar;

sal = new Producto(80005355, "Sal", 0.60);
azúcar = new Producto(800053588, "Azúcar", 0.81);
```

Agrupándolo todo, el código quedaría así:

```
public class Producto {
  int código;
  String descripción;
  double precio;

  // el constructor: inicializa el objeto Producto
  Producto(int código, String descripción, double precio) {
    this.código = código;
    this.descripción = descripción;
    this.precio = precio;
  }

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }

  public static void main(String[] args) {
    Producto sal, azúcar;

    sal = new Producto(80005355, "Sal", 0.60);
    azúcar = new Producto(80005388, "Azúcar", 0.81);
```

```
    System.out.println("Precio de 1 paquete de sal: " +
      sal.obtenerPrecio() + " EUR");
    System.out.println("Precio de 1 paquete de azúcar: " +
      azúcar.obtenerPrecio() + " EUR");
  }
}
```

Observad que lo que sigue al operador *new* no es otra cosa que la llamada al constructor de la clase. Después de la instanciación, los atributos de los objetos "sal" y "azúcar" ya estarán inicializados con los valores que hayamos pasado al constructor. El programa generará la salida siguiente:

```
Precio de 1 paquete de sal: 0.60 EUR
Precio de 1 paquete de azúcar: 0.81 EUR
```

Una clase puede tener más de un constructor, del mismo modo que puede tener dos o más funciones con el mismo nombre. Sin embargo, como en el caso de las funciones, los parámetros deben ser distintos.

Si no se define ningún constructor para una clase, ésta tendrá un constructor implícito: el constructor por defecto. El constructor por defecto, que no espera ningún parámetro, es el que usábamos en las primeras implementaciones de la clase "Producto", cuando aún no habíamos definido ningún constructor: !

Es necesario remarcar que el constructor se genera por defecto, en ausencia de constructores explícitos. Esto significa que, si existen constructores explícitos, estamos obligados a usar alguno de ellos para instanciar un objeto. La llamada al constructor por defecto deja de ser válida.

```
azúcar = new Producto(); // llamada al constructor por
                         // defecto
```

2.5. Extensión y herencia

La programación orientada a objetos (POO de ahora en adelante) establece un mecanismo fundamental que nos permite definir una clase en términos de otra: la extensión. La idea subyacente es partir de una clase general para generar una clase más específica que considere alguna característica o funcionalidad no cubierta por la clase más general.

Supongamos la clase "vehículo". Según el diccionario, un vehículo es todo aquello que sirve para transportar personas o cosas de un lugar a otro. Luego, podemos definir los atributos "número de plazas" y "carga máxima" para caracterizar a los vehículos. Sin embargo, hay vehículos con motor y sin motor. Es indudable que los primeros tendrán unos atributos ("poten-

cia", "consumo", etc.) de los que carecerán los segundos y viceversa ("tipo de tracción": animal, humana, etc.). Sin embargo, a la vez, ambos comparten los atributos propios de todo vehículo: "número de plazas" y "carga máxima".

Entonces, la POO nos permite definir las clases "vehículo con motor" y "vehículo sin motor" como clases derivadas (o hijas) de la clase "vehículo". Como clases hijas, heredan los atributos y los comportamientos de la clase madre ("número de plazas" y "carga máxima"), y pueden introducir otros nuevos más específicos ("potencia" y "consumo" en el caso de los vehículos con motor).

Supongamos ahora que en nuestro colmado empezamos a vender productos a granel. La aproximación anterior, en la que cada producto tenía un precio, deja de ser válida para todos los productos. Ahora hay productos que tienen un precio por kilogramo y un peso, además del código y la descripción. Está claro que, en gran medida, los productos que se venden a granel no difieren demasiado de los que se venden por unidades. De hecho, sólo cambia un poco el significado del concepto de precio, que ahora no es por unidad sino por kilo. Además, tenemos un nuevo atributo, el peso, y una nueva forma de computar el precio, el precio (por kilo) multiplicado por el peso.

Jugando con este símil, el código siguiente define la clase "ProductoGranel" como una extensión de la clase "Producto":

```
class ProductoGranel extends Producto {
  // añade el atributo peso (del producto)
  double peso;

  // el constructor también añade el parámetro peso
  ProductoGranel(int código, String descripción, double
    precio, double peso) {
    super(código, descripción, precio);
    this.peso = peso;
  }

  // para los productos vendidos a granel, el precio es
  // igual al resultado de multiplicar el peso (en Kg) por
  // el precio por Kg
  double obtenerPrecio() {
    return precio * peso;
  }
}
```

En primer lugar, observad el uso de la palabra clave *extends* (extiende):

```
class ProductoGranel extends Producto {
```

Mediante esta construcción, definimos una clase en términos de otra. Esto implica que la clase "ProductoGranel" tendrá los mismos atributos y funciones que la clase "Producto", más aquellos que defina de forma expresa. A este mecanismo de transmisión de atributos y funciones se le llama herencia.

Observad también la llamada al constructor de la clase "Producto" desde el constructor de la clase "ProductoGranel" mediante la palabra reservada *super* (superclase):

```
super(código, descripción, precio);
```

Como ya hemos comentado, la clase "ProductoGranel" hereda los atributos y las funciones de la clase "Producto". Esto le permite llamar al constructor de la clase "Producto" para inicializar aquellos atributos que ha heredado. La llamada al constructor de la clase madre, si la hay, debe ser la primera sentencia del constructor de la clase hija.

Finalmente, podemos observar que "ProductoGranel" define una función "obtenerPrecio", que es exactamente la misma que ya existía en la clase "Producto". Sin embargo, su código cambia sustancialmente: ahora el precio se calcula como el producto de los atributos "precio" (por kilo) y "peso":

```
double obtenerPrecio() {
  return precio * peso;
}
```

La nueva función sustituirá a la heredada de "Producto". Este mecanismo nos permite redefinir una función para que se ajuste a las características de la nueva clase. En el caso de la clase "ProductoGranel", este cambio era necesario, pues la función "obtenerPrecio" que había heredado simplemente devolvía el valor del atributo "precio".

El mecanismo de extensión nos permite utilizar objetos de la clase "ProductoGranel" como si fueran de la clase "Producto". El código siguiente imprime el nombre y el precio de varios productos independientemente de su tipo:

```
class Producto {
  int código;
```

```
  String descripción;
  double precio;

  // el constructor: inicializa el objeto Producto
  Producto(int código, String descripción, double precio) {
    this.código = código;
    this.descripción = descripción;
    this.precio = precio;
  }

  // fija el precio del producto
  void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  double obtenerPrecio() {
    return precio;
  }

  // devuelve la descripción del producto
  String obtenerDescripción() {
    return descripción;
  }
}

class ProductoGranel extends Producto {
  // añade el atributo peso (del producto)
  double peso;

  // el constructor también añade el parámetro peso
  ProductoGranel(int código, String descripción,
    double precio, double peso) {
    super(código, descripción, precio);
    this.peso = peso;
  }

  // para los productos vendidos a granel, el precio es
  // igual al resultado de multiplicar el peso (en Kg) por
  // el precio por Kg
  double obtenerPrecio() {
    return precio * peso;
  }
}
```

```
public class Caja {
  // muestra por pantalla el precio de un producto
  public static void escribirPrecio(Producto p) {
    System.out.println(p.obtenerDescripción() + " " +
      p.obtenerPrecio() + " EUR");
  }

  public static void main(String[] args) {
    Producto sal;
    ProductoGranel mango, salmón;

    // crea los productos sal, salmón y mango
    sal = new Producto(80005355, "Sal", 0.60);
    salmón = new ProductoGranel(80005373, "Salmón",
      9.55, 0.720);
    mango = new ProductoGranel(80005312, "Mango", 2.99,
      0.820);

    // escribe el precio de los tres productos
    escribirPrecio(sal);
    escribirPrecio((Producto)salmón);
    escribirPrecio((Producto)mango);
  }
}
```

Como podemos observar, en la llamada a la función “escribirPrecio” para los objetos de la clase “ProductoGranel” hay una peculiaridad: el uso del operador de conversión:

```
escribirPrecio((Producto)mango)
```

El operador de conversión permite cambiar el tipo de una variable a otro tipo compatible (*casting*, en inglés). En el ejemplo, forzamos a que “salmón” y “mango” se traten como si fuesen de la clase “Producto”. El operador de conversión toma la forma del tipo destino encerrado entre paréntesis justamente delante de la variable que deseamos convertir.

El resultado será el siguiente:

```
Sal 0.60 EUR
Salmón 6.876 EUR
Mango 2.4518 EUR
```

2.6. Los paquetes y la directiva *import*

Para estructurar el código y evitar conflictos con los nombres de las clases, el lenguaje Java nos permite agrupar las clases en paquetes (*packages*).

Normalmente, los paquetes agrupan clases que están relacionadas por alguna funcionalidad o característica. Por ejemplo, se suelen crear paquetes para agrupar las clases que implementan la interfaz de usuario de una aplicación o aquellas que proporcionan algún tipo de cálculo matemático o probabilístico.

La agrupación de clases en paquetes nos permite:

- Poner de manifiesto la relación entre un conjunto de clases.
- Identificar un conjunto de clases con una funcionalidad.
- Evitar los conflictos con los nombres de las clases: puede haber clases homónimas en paquetes distintos.

Para crear un paquete, basta con escribir en la parte superior de un fichero fuente la palabra *package* seguida del nombre del paquete. Por ejemplo, las líneas siguientes:

```
package postgradosig;
class Asignatura {
  // (...)
```

definen una clase llamada "Asignatura" perteneciente al paquete "postgradosig".

Los nombres de los paquetes tienen las mismas restricciones que los nombres de las variables: no pueden comenzar con un carácter numérico, no pueden llamarse igual que un elemento del lenguaje y no pueden contener espacios.

Los paquetes se pueden dividir en subpaquetes de manera arbitraria, formando una jerarquía de paquetes. Las líneas siguientes:

```
package edu.uoc.postgradosig;
class Asignatura {
  // (...)
}
```

definen una clase llamada "Asignatura" perteneciente al paquete "postgradosig", que se encuentra dentro del paquete "uoc", que a su vez se halla dentro del paquete "edu". Como se puede observar, los paquetes se separan mediante el carácter punto (".").

Cada paquete constituye una unidad léxica independiente dentro del código. Esto significa que los nombres de las clases no traspasan los límites del paquete, lo que permite definir clases homónimas en paquetes distintos. La inde-

pendencia de los paquetes es tal que desde un paquete no se pueden *ver* las clases de otros paquetes, al menos directamente.

Cuando desde una clase se desea acceder a otra clase situada en otro paquete, es necesario indicarlo de forma explícita mediante la directiva *import* (importar). De hecho, esta directiva no se hace otra cosa que importar la definición de la clase.

Supongamos la clase "Alumno" perteneciente al paquete "edu.uoc.postgradosig.alumno" y la clase "Asignatura" perteneciente al paquete "edu.uoc.postgradosig.asignatura". Supongamos también que la clase "Asignatura" tiene una función "matricular" que permite matricular a un alumno a la asignatura. Entonces, el código de la clase "Asignatura" sería parecido al siguiente:

```
package edu.uoc.postgradosig.asignatura;

// importa la clase "Alumno"
import edu.uoc.postgradosig.alumno.Alumno;

class Asignatura {
  // (...)

  public void matricular(Alumno alumno) {
    // (...)
  }
}
```

Observad que a la palabra reservada *import* le sigue el nombre cualificado de la clase, es decir, el nombre del paquete más el nombre de la clase. Es importante remarcar que la directiva *import* no hubiera sido necesaria si las clases hubiesen pertenecido al mismo paquete.

2.7. Visibilidad y encapsulación

Hasta ahora hemos accedido a los atributos y las funciones de un objeto libremente, sin ninguna restricción. Sin embargo, como los atributos contienen el estado de un objeto, no suele ser deseable que código ajeno a la clase pueda acceder a ellos y modificarlos, porque, al desconocer el papel que desempeñan cada uno de los atributos en la implementación de la clase, esto podría dejar el objeto en un estado erróneo.

Se llama encapsulación a la técnica consistente en ocultar el estado de un objeto, es decir, sus atributos, de manera que sólo pueda cambiarse mediante un conjunto de funciones definidas a tal efecto.

Los atributos y las funciones de una clase pueden incorporar en su definición un modificador que especifique la visibilidad de dicho elemento. Hay tres modificadores explícitos en Java:

* ***public***: el elemento es público y puede accederse a él desde cualquier clase,
* ***protected***: el elemento es semiprivado y sólo pueden acceder a él la clase que lo define y las clases que lo extienden, y
* ***private***: el elemento es privado y sólo puede acceder a él la clase que lo define.

Si no se especifica ningún nivel de visibilidad, se aplica la regla de visibilidad implícita (conocida como *package*): el atributo o función será público para todas las clases del mismo paquete, y privado para las demás.

A modo de ejemplo, veamos una nueva versión de la clase "Producto":

```
class Producto {
  private int código;
  private String descripción;
  private double precio;

  // el constructor: inicializa el objeto Producto
  public Producto(int código, String descripción,
    double precio) {
    this.código = código;
    this.descripción = descripción;
    this.precio = precio;
  }

  // fija el precio del producto
  public void fijarPrecio(double precioNuevo) {
    precio = precioNuevo;
  }

  // devuelve el precio del producto
  public double obtenerPrecio() {
    return precio;
  }

  // devuelve la descripción del producto
  public String obtenerDescripción() {
    return descripción;
  }
}
```

Como se puede apreciar, se han declarado todos los atributos privados y las funciones públicas. Esto implica que la sentencia:

```
sal.precio = 0.60;
```

en la que "sal" es una instancia de la clase "Producto", será inválida. En su lugar, debemos usar una llamada a la función "cambiarPrecio" creada a tal efecto:

```
sal.cambiarPrecio(0.60);
```

Es importante remarcar que al menos un constructor de la clase debe ser público. En caso contrario, no se va a poder instanciar ningún objeto de la clase.

3. La API de Java

El lenguaje Java, en su distribución estándar, viene acompañado de una extensa API* (de *Application Programming Interface*, 'interfaz de programación de aplicaciones') que proporciona un conjunto de clases ya implementadas que facilitan o resuelven los problemas más habituales con los que se encuentra un programador.

*** La API puede consultarse en: http://download.oracle.com/javase/1.5.0/docs/api/**

Las clases de la API de Java, que estan estructuradas en paquetes, ofrecen, entre otras, las siguientes funcionalidades:

- lectura y escritura de ficheros en el paquete "java.io",
- representación de gráficos: dibujo de líneas, polígonos, elipses, etc., en el paquete "java.awt",
- creación de interfaces gráficas de usuario en el paquete "javax.swing", y
- gestión y ordenación de objetos mediante pilas, colas, listas, etc., en el paquete "java.util" .

Todas estas clases se pueden importar en nuestro código mediante la directiva *import*. Por ejemplo, el código siguiente hace uso de la clase "Vector" para implementar una cola de tareas que se deben realizar:

```
import java.util.Vector; // importamos la clase Vector

class Tarea {
 private String descripción; // descripción de la tarea

 // constructor de la clase Tarea: su único parámetro es
 // la descripción (textual) de la tarea
 public Tarea(String desc) {
  descripción = desc;
 }

 // devuelve la descripción de la tarea
 public String obtDescripción() {
  return descripción;
 }

 public static void main(String[] args) {
  // el Vector "tareas" almacenará las tareas en orden
  Vector tareas;
  Tarea tarea;
```

```
    // crea una cola de tareas
    tareas = new Vector();

    // agrega tareas a la cola
    tareas.add(new Tarea("Barrer"));
    tareas.add(new Tarea("Fregar"));
    tareas.add(new Tarea("Hacer la compra"));

    // recupera las tareas el mismo orden que fueron
    // agregadas

    // mientras la cola no esté vacía...
    while (! tareas.isEmpty()) {
     // saca el primer elemento (el 0) de la cola...
     tarea = (Tarea)tareas.remove(0);
     // ... y lo muestra por pantalla
     System.out.println(tarea.obtDescripción());
    }
  }
}
```

En el web de la API de Java se puede encontrar la documentación completa de la clase “Vector”*. Entre otras cosas, se explica el funcionamiento de las funciones “remove” e “isEmpty”.

*** http://download.oracle.com/javase/1.5.0/docs/api/java/util/Vector.html**

Resumen

En este módulo nos hemos introducido en la programación orientada a objetos de la mano del lenguaje Java. En primer lugar, hemos estudiado los mecanismos comunes a todos los lenguajes imperativos y procedimentales (asignación de variables, estructuras de control de flujo y llamadas a funciones) para continuar con los propios de los lenguajes de orientación a objetos (definición de clase y objeto, extensión y herencia).

Además, se ha detallado la razón y el funcionamiento de las estructuras de control de flujo primordiales, y hemos aprendido a realizar recorridos y búsquedas. Todo esto, combinado con el uso de funciones, nos permitirá estructurar correctamente un código.

Para finalizar, aunque este módulo se ha centrado exclusivamente en el lenguaje Java, las técnicas aquí aprendidas nos permitirán introducirnos con relativa facilidad a otros lenguajes de programación parecidos, como Ada, C, C++, etc.

Bibliografía

Oracle, "*The Java(tm) Tutorials*", http://download.oracle.com/javase/tutorial/.

Oracle, "*Java(tm) 2 Platform Standard Edition 5.0 API Specification*", http://download.oracle.com/javase/1.5.0/docs/api/.

Adaptación y extensión de un SIG

Albert Gavarró Rodríguez

PID_00174756



Índice

Introducción

En el módulo anterior, nos hemos introducido en el mundo de la programación orientada a objetos de la mano del lenguaje Java. El objetivo del módulo era claro: proporcionar los conocimientos mínimos necesarios para realizar pequeñas implementaciones para, posteriormente, abordar con mínimas garantías la programación de un SIG.

Introducidos ya en el mundo de la programación, en este módulo aprenderemos a implementar un SIG de escritorio o –lo que es lo mismo– un SIG que se ejecuta en la máquina del usuario de la misma forma en que lo hace cualquier otra aplicación.

Como la programación desde cero de un SIG es muy compleja y nos llevaría muchas horas de dedicación (en caso de que poseyéramos los suficientes conocimientos teóricos y técnicos como para llevarla a cabo), lo cual no es nuestro objetivo, usaremos como punto de partida un SIG ya implementado, que adaptaremos a nuestras necesidades mediante el desarrollo de complementos. El SIG del que partiremos será gvSIG.

Objetivos

Una vez finalizado el módulo, deberíamos ser capaces de:

1. Extender gvSIG mediante complementos para adaptarlo a nuestras necesidades.

2. Agregar capas vectoriales a una vista de gvSIG.

3. Señalar lugares del mapa, trazar rutas o delimitar áreas mediante el uso de capas vectoriales.

4. Transformar las coordenadas del mapa en coordenadas geográficas y viceversa.

5. Determinar los límites visibles del mapa (*viewport*).

6. Implementar herramientas que interactúen con el mapa.

1. gvSIG

gvSIG es una herramienta orientada al manejo de información geográfica. Permite acceder a información vectorial y rasterizada georreferenciada, así como a servidores de mapas que cumplan las especificaciones que impone el OGC (*Open Geospatial Consortium* —Consorcio Abierto Geoespacial—).

Las razones para usar gvSIG son variadas:

- Está programado en Java, lo que significa que puede ejecutarse virtualmente en casi cualquier máquina.
- Es software libre*, lo que nos da la libertad de modificarlo y distribuirlo a nuestro antojo. El coste de este software es cero.
- Su funcionalidad se puede ampliar fácilmente mediante complementos que se pueden programar en Java o en Jython**.
- Está probado por una base amplia de usuarios, lo que asegura una buena estabilidad del código.

*** Está sometido a una licencia GNU GPL 2.0. Se pueden leer los términos de la licencia en: http://www.gnu.org/licenses/gpl-2.0.html**

**** Jython es una implementación del lenguaje Python para la plataforma Java: http://www.jython.org/**

1.1. Arquitectura de gvSIG

gvSIG se estructura alrededor de un núcleo de funcionalidad relativamente pequeño. A este núcleo, que contiene las partes esenciales del sistema, se le pueden añadir complementos para dotarlo de nuevas funcionalidades. Esta arquitectura permite descomponer el sistema en partes pequeñas, más fáciles de implementar y mantener. Actualmente, gran parte de la distribución de gvSIG son complementos.

Aunque los complementos pueden ser de naturaleza muy variopinta, siempre se muestran al usuario de forma homogénea. Es decir, el aspecto gráfico y la forma de trabajar con un complemento u otro serán más o menos parecidos.

Los complementos se integran en el marco de gvSIG mediante las llamadas “extensiones”, que no son más que un conjunto de clases Java que añaden nuevas funcionalidades a la aplicación.

El núcleo de gvSIG está formado por tres subsistemas:

- ***FMap.*** Agrupa la lógica SIG. Contiene las clases que se encargan de generar los mapas, gestionar las capas, transformar coordenadas y realizar búsquedas, consultas, análisis, etc.

- ***gvSIG***. Agrupa las clases encargadas de gestionar la interfaz gráfica de la aplicación y representar en pantalla los mapas generados por el subsistema *FMap*. En otras palabras, este subsistema se encarga de proporcionar una interfaz de usuario que proporciona acceso a las funcionalidades de gvSIG.

- ***Subdriver***. Contiene las clases que se encargan de dar acceso a los datos, tanto locales como remotos (por ejemplo, servicios OGC). Además, permite escribir datos geográficos en los formatos más comunes.

Para llevar a cabo su cometido, un complemento puede acceder a uno o más subsistemas de gvSIG.

1.2. Anatomía de un complemento

Como mínimo, un complemento está constituido por:

- un fichero XML que describe el complemento, y
- el código del complemento.

El fichero XML, que debe llamarse *config.xml*, proporciona a gvSIG la información necesaria sobre el complemento para que pueda cargarlo e integrarlo en la interfaz gráfica. Entre otras cosas, especifica el directorio que contiene el código del complemento, las dependencias que tiene con otros complementos y los elementos de interfaz de usuario (menús, barras de herramientas, etc.) que añade a la interfaz de gvSIG.

El código del complemento es un conjunto de clases Java que implementan la funcionalidad aportada por el complemento. Probablemente, el código irá acompañado de ficheros de configuración, imágenes, datos geográficos o cualquier otro dato que sea necesario para la ejecución de éste.

A lo largo de este módulo, vamos a implementar, a modo de ejemplo, un complemento llamado "Construcciones" que mostrará elementos constructivos (edificios y caminos) sobre una base topográfica. En los apartados siguientes, aprenderemos a crear un complemento y, por medio de él, a:

- agregar una capa de información geográfica a una vista,
- dibujar líneas y polígonos georeferenciados sobre un mapa, y
- crear una herramienta que muestre información sobre los elementos dibujados.

1.3. El fichero *config.xml*

El fichero *config.xml* contiene los datos de configuración del complemento. Este fichero, que obligatoriamente debe acompañar al código, define básica-

mente los elementos de interfaz de usuario que se van a añadir a la aplicación (herramientas, menús y controles de la barra de estado), a la vez que los asocia con clases del código. Cada una de estas asociaciones es una "extensión".

El fichero *config.xml* debe escribirse en lenguaje XML (de *eXtensible Markup Language* —'lenguaje de marcas extensible'—) y ha de seguir unas pautas determinadas.

Un documento XML está constituido por elementos que forman una jerarquía. Estos elementos se denotan mediante el uso de etiquetas de inicio y fin. Las etiquetas tienen la forma <*nombre*>, </*nombre*> o <*nombre* />, en donde *nombre* es el nombre del elemento. La primera es una etiqueta de inicio, la segunda de fin, y la última una combinación de ambas. Las dos primeras se utilizan cuando un elemento puede contener otros elementos, y sirven para marcar el inicio y el fin de dicho elemento. La última está reservada a elementos que no contienen otros. Además, los elementos pueden tener atributos cuyos valores se establecen de forma muy parecida a las asignaciones de los lenguajes de programación.

Veamos el fichero de configuración que usará nuestro complemento:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<plugin-config>
  <depends plugin-name="com.iver.cit.gvsig" />
  <libraries library-dir="."/>
  <extensions>
    <extension class-name=
      "edu.uoc.postgradosig. construcciones.Construcciones"
      description="Complemento que dibuja construcciones"
      active="true"
      priority="50">
      <menu text="Postgrado SIG/Construcciones"
       action-command="MENU_CONSTRUCCIONES" />
    </extension>
  </extensions>
</plugin-config>
```

La primera línea, llamada prólogo, describe la versión del lenguaje XML utilizada para escribir el documento y su codificación. Como el prólogo no forma parte de la configuración del complemento, no debemos preocuparnos demasiado por su contenido. Simplemente, siempre utilizaremos la misma fórmula.

El significado del resto de elementos es el siguiente:

- ***plugin-config***. Es una etiqueta que engloba todas las opciones de configuración. Es la raíz de la jerarquía de elementos.

- ***depends***. Enumera los complementos de los que depende nuestro complemento para funcionar correctamente. El nombre del complemento se especifica mediante el atributo "plugin-name". Debe haber un elemento "depends" por cada complemento. Un complemento depende de otro cuando el código del primero utiliza alguna de las clases del código del segundo. Normalmente, todos los complementos dependerán del complemento gvSIG ("com.iver.cit.gvsig"), que es la aplicación en sí.

- ***libraries***. Especifica el directorio en el que reside el código de nuestro complemento. El nombre del directorio se especifica mediante el atributo "library-dir". El punto como nombre de directorio significa que el código se encuentra en el mismo directorio que el fichero de configuración.

- ***extensions***. Marca el inicio de la lista de extensiones del complemento.

- ***extension***. Declara una extensión de gvSIG. La declaración incluye el nombre de la clase Java que implementa la extensión y una lista de elementos de interfaz de usuario que deben añadirse a la aplicación y que están asociados con la extensión. El elemento "extension" tiene los atributos siguientes:

  – ***class-name***: especifica el nombre de la clase que implementa la extensión.

  – ***description***: describe la funcionalidad que aporta la extensión. Esta etiqueta es meramente informativa.

  – ***active***: especifica si la extensión está activa o inactiva. Si el valor de este atributo es "false", gvSIG no cargará la extensión.

  – ***priority***: establece la prioridad de carga de la extensión respecto al resto de extensiones declaradas en el fichero de configuración. Cuanto menor sea el número, antes se cargará la extensión. En nuestro caso, sólo hay una extensión, por lo que el valor de la prioridad no es fundamental.

- ***menu***. Define una nueva entrada del menú de gvSIG y la asocia a la extensión. El elemento "menu" tiene dos atributos:

  – ***text***: establece el nombre y la ubicación del menú. Puede usarse la barra ("/") para establecer diversos niveles de menú.

  – ***action-command***: especifica el identificador de la acción. Cada vez que se active el menú, se llamará a la clase asociada pasándole el valor de este atributo. Esto nos permite asociar varios menús a una sola clase.

En el ejemplo, se declara la opción de menú *Postgrado SIG* → *Construcciones* con el identificador asociado "MENU_CONSTRUCCIONES". El resultado será el siguiente:



1.4. La clase "Extensión"

Una vez creado el fichero XML de configuración de la extensión, en cuyo contenido hemos definido la clase que la implementa y los elementos de interfaz de usuario que nos permitirán acceder a su funcionalidad, llega el momento de implementarla.

Para ser consideradas como tales, las extensiones deben extender (valga la redundancia) la clase "Extension"* y sobrecargar sus funciones. Al reconocer la clase como una extensión, gvSIG se comunicará con ella por medio de estas funciones, que actuarán a modo de interfaz entre gvSIG y la extensión.

*** La documentación de la clase se puede consultar en: API gvSIG/com/iver/andami/plugins/Extension.html**

A continuación, se muestra una posible implementación de la extensión "Construcciones":

```
package edu.uoc.postgradosig.construcciones;

import com.iver.andami.plugins.Extension;
import javax.swing.JOptionPane;

public class Construcciones extends Extension {
  public void initialize() {
    // aquí se realizan las tareas de inicialización
  }

  public boolean isEnabled() {
    return true; // está habilitada
  }
```

```
  public boolean isVisible() {
    return true; // es visible
  }

  public void execute(String actionCommand) {
    // muestra un mensaje por pantalla
    JOptionPane.showMessageDialog(null,
      "Esta extensión muestra elementos constructivos.");
  }
}
```

Como se puede observar, la clase "Construcciones" extiende la clase "Extension" e implementa (sobrecarga) cuatro funciones: *initialize*, *isEnabled*, *isVisible* y *execute*. El cometido de cada una de ellas es el siguiente:

- ***void initialize().*** Se utiliza para inicializar la extensión. En esta función emplazaremos llamadas a funciones que se encarguen de reservar recursos, leer configuraciones, etc. Debemos fijarnos en que esta función desempeña un papel muy parecido al del constructor de la clase. Sin embargo, por coherencia con gvSIG, reservaremos las tareas más arduas para esta función, limitando el cometido del constructor —si lo hay— a la inicialización de variables.

- ***boolean isEnabled().*** Indica si la extensión debe considerarse habilitada o no. Cuando una extensión no está habilitada, los elementos de interfaz de usuario (botones, menús o controles de la barra de estado) que tiene asociados no son accesibles. En la mayoría de sistemas, estos elementos se mostrarán en tonos grisáceos. gvSIG llama periódicamente a esta función respondiendo a los eventos de la interfaz gráfica.

- ***boolean isVisible().*** Indica si los elementos de interfaz de usuario que tiene asociados la extensión deben estar visibles o no. Si esta función devuelve *false*, no se mostrará ninguno de sus elementos. gvSIG llama periódicamente a esta función respondiendo a los eventos de la interfaz gráfica.

- ***void execute(String actionCommand).*** gvSIG llama a esta función cada vez que se activa la extensión, es decir, cada vez que el usuario accede a alguno de los elementos de interfaz de usuario que la extensión ha definido. El parámetro "actionCommand" contiene el valor del atributo "action-command" (definido en el fichero de configuración) del elemento al que se ha accedido.

Todas las extensiones están obligadas a implementar estas cuatro funciones. Al tratarse de funciones sobrecargadas, debemos prestar especial atención a respetar tanto los modificadores de visibilidad como el tipo de la función y de los parámetros. Opcionalmente, si se desea un mayor control sobre los proce-

sos de inicialización y descarga de la extensión, también se pueden implementar las funciones siguientes:

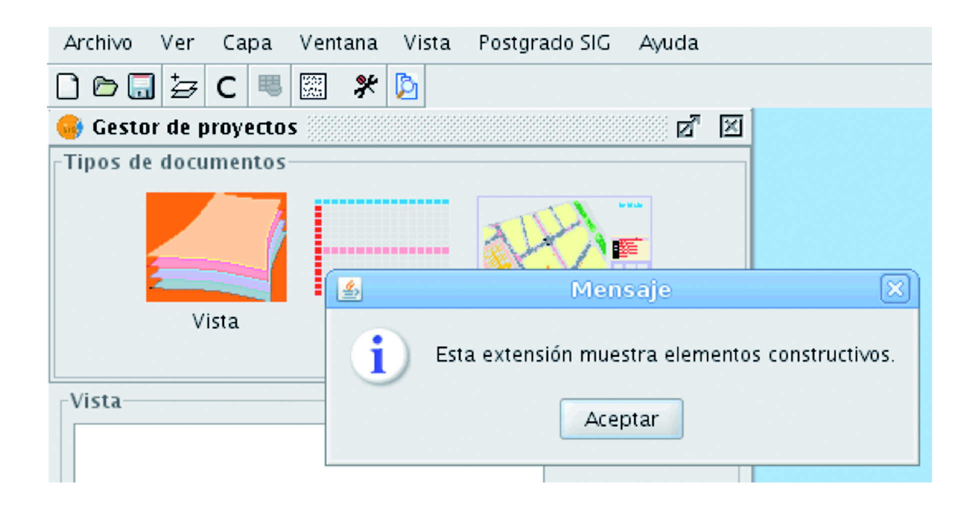
- ***void postInitialize()***. Se llama a esta función después de haberse llamado a la función *initialize* de todas las extensiones. Se utilizará para realizar tareas de inicialización que requieran una interacción con otras extensiones.
- ***void terminate()***. Se llama a esta función al terminar la ejecución de gvSIG. Típicamente, en esta función almacenaremos las preferencias del usuario, liberaremos recursos, etc.

Volvamos al ejemplo: la extensión mostrará una ventana con un mensaje descriptivo cada vez que se active. Este código, combinado con el fichero de configuración que ya habíamos definido, dará lugar a que se muestre el mensaje cada vez que se seleccione la opción de menú *Postgrado SIG* → *Construcciones*. Cabe destacar que, en este caso, el contenido del parámetro "actionCommand" será "MENU_CONSTRUCCIONES", tal y como habíamos definido en el fichero de configuración. También es importante observar que las funciones *isEnabled* e *isVisible* devuelven *true*, lo que indica en todo momento que la extensión está habilitada y es visible.

Debemos observar también que el mensaje informativo se muestra mediante la función *showMessageDialog* de la clase "JOptionPane"*.

* http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html

El resultado será el siguiente:



1.5. Servicios prestados a las extensiones

Hasta ahora hemos visto cómo gvSIG se comunica con las extensiones para determinar su estado o acceder a su funcionalidad, pero hemos ignorado cómo las extensiones acceden al contexto de gvSIG. Entendemos como contexto todas aquellas clases y objetos que representan las ventanas, las vistas, los mapas y los demás elementos que conforman el entorno de ejecución de gvSIG. El acceso a este contexto se logra mediante la clase "PluginServices"*.

* Todas las clases que componen la versión 1.1 de la API gvSIG están explicadas en detalle en la documentación de la API que se entrega conjuntamente con estos materiales. La página principal de la documentación se encuentra en: API gvSIG/index.html

Si continuamos con la implementación de nuestra extensión, es deseable que la extensión (y, en consecuencia, la opción del menú) sólo esté habilitada si se

trabaja sobre una vista. Para conseguir este efecto, utilizaremos la clase "PluginServices" para obtener la ventana activa, comprobaremos si se trata de una ventana de vista e informaremos a gvSIG sobre si está habilitada o no la extensión mediante la función *isEnabled*:

```
package edu.uoc.postgradosig.construcciones;

import com.iver.andami.plugins.Extension;
import javax.swing.JOptionPane;
import com.iver.andami.PluginServices;
import com.iver.cit.gvsig.project.documents.view.gui.View;
import com.iver.andami.ui.mdiManager.IWindow;

public class Construcciones extends Extension {

  public void initialize() {
  }

  public boolean isVisible() {
    return true; // es visible
  }

  public boolean isEnabled() {
    IWindow ventanaActiva;

    // obtiene la ventana activa
    ventanaActiva = PluginServices.getMDIManager().
      getActiveWindow();

    // estará habilitada sólo si la ventana activa es una
    // ventana de vista
    return ventanaActiva instanceof View;
  }

  public void execute(String actionCommand) {
    // muestra un mensaje por pantalla
    JOptionPane.showMessageDialog(null,
      "Esta extensión muestra elementos constructivos.");
  }
}
```

Como se puede observar, utilizamos la clase "PluginServices" para obtener, en primer lugar, el gestor de ventanas de gvSIG*, y, después, la ventana activa, mediante las funciones *getMDIManager* y *getActiveWindow* respectivamente:

*** Un gestor de ventanas es un software que se encarga de gestionar un entorno de ventanas. Proporciona los mecanismos necesarios para crear, alterar, ordenar, presentar y destruir ventanas, entre otros.**

```
ventanaActiva = PluginServices.getMDIManager().
  getActiveWindow();
```

La ventana activa es de la clase “IWindow”. De hecho, todas las ventanas de gvSIG extienden esta clase. Por ejemplo, las clases “View” y “Table” extienden “IWindow” e implementan las ventanas de vista y de tabla respectivamente.

Obtenida la ventana activa, sólo nos queda determinar si se trata de una ventana de vista o no. Esto lo hacemos mediante el operador *instanceof*, que nos permite comprobar si un objeto es instancia de una clase determinada*. Como ya hemos visto, el objeto “ventanaActiva” será instancia de la clase “View” si se trata de una ventana de vista. En consecuencia, la extensión estará activa sólo si “ventanaActiva” es una instancia de la clase “View”:

*** Dada una clase *C* y un objeto *o*, *o instanceof C* devolverá *true* si *o* es una instancia de *C* o de alguna de las clases que extienden *C*. En cualquier otro caso se devolverá *false*.**

```
return ventanaActiva instanceof View;
```

La clase “PluginServices” proporciona otras muchas funciones que dan soporte a las extensiones. A continuación, comentamos las más relevantes:

- ***String[] getArguments().*** Devuelve un vector con los parámetros de arranque de la aplicación.

- ***String getArgumentByName(String name).*** Devuelve el valor del parámetro de arranque especificado. Para que funcione correctamente, el parámetro debe tener la forma “-<nombre_parámetro>=<valor>”.

- ***Logger getLogger().*** Devuelve un objeto “Logger” mediante el que se pueden escribir trazas en el fichero de *log** de gvSIG. El objeto proporciona (entre otras) tres funciones (*error*, *warn* e *info*) que permiten escribir distintos tipos de mensajes según su criticidad (de error, de aviso y de información respectivamente).

*** Un fichero de *log* es aquel que almacena información sobre los sucesos que ocurren en un sistema. Es como un cuaderno de bitácora en el que se escriben evidencias que más tarde servirán par detectar posibles anomalías en el software.**

- ***MainFrame getMainFrame().*** Devuelve el objeto que representa a la ventana principal de gvSIG. Este objeto se utiliza principalmente para cambiar la herramienta activa.

- ***MDIManager getMDIManager().*** Devuelve un objeto de la clase “MDIManager” que representa al gestor de ventanas de gvSIG. Utilizaremos el gestor cuando queramos añadir nuevas ventanas o gestionar las existentes.

- ***void setPersistentXML(XMLEntity entity).*** Permite almacenar información sobre el estado de la extensión que podrá ser recuperada más tarde. Puede ser interesante guardar información de estado al finalizar la sesión de gvSIG.

- ***XMLEntity getPersistentXML().*** Devuelve un objeto “XMLEntity” que contiene el último estado guardado de la extensión. Es la función complementaria de “setPersistentXML”.

1.6. Agregar una capa vectorial

Para representar elementos constructivos sobre un mapa, utilizaremos líneas y polígonos. Las líneas nos permitirán dibujar caminos, y los polígonos, edificios.

El primer paso será crear un mapa vectorial en memoria. Una vez creado, dibujaremos sobre él los elementos constructivos, que previamente habremos georeferenciado, y les asociaremos datos. Terminado el mapa, lo sobrepondremos a una base topográfica mediante el uso de capas y obtendremos así un nuevo mapa.

1.6.1. Crear una capa vectorial en memoria

Para crear un mapa vectorial en memoria, usaremos la clase "ConcreteMemoryDriver". Las líneas siguientes:

```
ConcreteMemoryDriver mapa;

mapa = new ConcreteMemoryDriver();
mapa.setShapeType(FShape.LINE + FShape.POLYGON);
mapa.getTableModel().setColumnIdentifiers(new String[] {
  "ID", "Descripcion"});
```

crean un mapa vectorial llamado "mapa". Además, se establecen, mediante la función *setShapeType*, los tipos de figuras que contendrá, "FShape.LINE" (por líneas) y "FShape.POLYGON" (por polígonos), y mediante la función *setColumnIdentifiers*, los atributos asociados a cada una de ellas, "ID" y "Descripcion". Los atributos se pueden definir arbitrariamente según nuestras necesidades.

1.6.2. Dibujar líneas y polígonos

Una vez creado el mapa, podemos empezar a añadirle figuras. Supongamos que, en una fase previa de captación de datos, hemos localizado una vivienda y un camino cuyos vértices son los siguientes:

Objeto	**Vértices (UTM 31N/ED50)**	
	x	***y***
Vivienda	356.270,7	4.727.507,1
	356.273,0	4.727.503,3
	356.278,5	4.727.505,8
	356.275,5	4.727.509,8
Camino	356.242,7	4.727.498,8
	356.268,0	4.727.509,8
	356.281,0	4.727.519,1

La vivienda la dibujaremos mediante el objeto "FPolygon2D". El código siguiente:

```
GeneralPathX ruta;
FPolygon2D edificio;

// definimos la geometría del edificio
ruta = new GeneralPathX();
ruta.moveTo(356270.7, 4727507.1); // mueve el cursor al inicio
ruta.lineTo(356273.0, 4727503.3); // dibuja el primer segmento
ruta.lineTo(356278.5, 4727505.8); // dibuja el segundo segmento
ruta.lineTo(356275.5, 4727509.8); // dibuja el tercer segmento
ruta.lineTo(356270.7, 4727507.1); // dibuja el cuarto segmento

// creamos el objeto edificio a partir de la geometría definida
// previamente
edificio = new FPolygon2D(ruta);

// agregamos el edificio al mapa
mapa.addShape(edificio, new Object[] {
  // valor del atributo "ID"
  ValueFactory.createValue(1),
  // valor del atributo "Descripcion"
  ValueFactory.createValue("Casa unifamiliar.") });
```

dibuja la vivienda y la agrega al mapa. Como se puede observar, primero usamos un objeto de la clase "GeneralPathX" para definir la geometría de la figura, y después creamos el polígono mediante la clase "FPolygon2D". En un último paso, lo agregamos al mapa y especificamos el valor de sus atributos. Observad que la función *createValue* está sobrecargada, lo que permite pasarle por parámetro tanto enteros como texto. También debemos notar que trabajamos directamente con coordenadas geográficas.

El camino lo dibujaremos de forma muy parecida. La única diferencia es que utilizaremos la clase "FPolyline2D" en lugar de "FPolygon2D":

```
GeneralPathX ruta;
FShape camino;

// definimos la geometría del edificio
ruta = new GeneralPathX();
ruta.moveTo(356242.7, 4727498.8);
ruta.lineTo(356268.0, 4727509.8);
ruta.lineTo(356281.0, 4727519.1);
```

```
// creamos el objeto camino a partir de la geometría
// definida previamente
camino = new FPolyline2D(ruta);

// agregamos el camino al mapa
mapa.addShape(camino, new Object[] {
     ValueFactory.createValue(2),
     ValueFactory.createValue("Camino de herradura.") });
```

1.6.3. Crear una capa

Una vez dibujado el mapa, ya sólo nos queda agregarlo a la vista actual como una capa. Esta tarea se reparte entre las clases “LayerFactory”, que crea la capa, y “MapControl”, que la agrega a la vista actual:

```
FLayer capa;
MapControl controlMapa;
View vistaActual;

// obtiene la ventana de vista activa
vistaActual = (View)PluginServices.getMDIManager().
  getActiveWindow();

// obtiene el objeto que controla el mapa
controlMapa = vistaActual.getMapControl();

// crea una capa a partir de un mapa
capa = LayerFactory.createLayer("Construcciones", mapa,
  controlMapa.getProjection());

// hace visible la capa
capa.setVisible(true);

// agrega la capa a la vista actual
controlMapa.getMapContext().getLayers().addLayer(capa);
```

Como se puede observar, la capa se crea mediante la función *createLayer* de la clase “LayerFactory”. Esta función espera tres argumentos: el nombre de la capa (“Construcciones”), el mapa que contendrá y la proyección con la que se representará (la misma que se utiliza en la vista). En un segundo paso, se agrega la capa a la vista mediante la función *addLayer*. Como podemos ver, para poder llamar a la función necesitamos obtener, en primer lugar, el contexto del mapa (“getMapContext”), y en segundo lugar, la colección de las capas que contiene (“getLayers”).

Recapitulando, el código de la extensión nos queda así:

```
package edu.uoc.postgradosig.construcciones;

import com.hardcode.gdbms.engine.values.ValueFactory;
import com.iver.andami.PluginServices;
import com.iver.andami.plugins.Extension;
import com.iver.andami.ui.mdiManager.IWindow;
import com.iver.cit.gvsig.fmap.MapControl;
import com.iver.cit.gvsig.fmap.core.FPolygon2D;
import com.iver.cit.gvsig.fmap.core.FPolyline2D;
import com.iver.cit.gvsig.fmap.core.FShape;
import com.iver.cit.gvsig.fmap.core.GeneralPathX;
import com.iver.cit.gvsig.fmap.drivers.
  ConcreteMemoryDriver;
import com.iver.cit.gvsig.fmap.layers.FLayer;
import com.iver.cit.gvsig.fmap.layers.LayerFactory;
import com.iver.cit.gvsig.project.documents.view.gui.View;

public class Construcciones extends Extension {

  public void execute(String actionCommand) {
    ConcreteMemoryDriver mapa;
    GeneralPathX ruta;
    FShape edificio, camino;
    FLayer capa;
    MapControl controlMapa;
    View vistaActual;

    // crea un mapa vectorial en memoria
    mapa = new ConcreteMemoryDriver();
    mapa.setShapeType(FShape.LINE + Fshape.POLYGON);
    mapa.getTableModel().setColumnIdentifiers(
      new String[] { "ID", "Descripcion"});

    // definimos la geometría del edificio
    ruta = new GeneralPathX();
    ruta.moveTo(356270.7, 4727507.1);
    ruta.lineTo(356273.0, 4727503.3);
    ruta.lineTo(356278.5, 4727505.8);
    ruta.lineTo(356275.5, 4727509.8);
    ruta.lineTo(356270.7, 4727507.1);

    // creamos el objeto edificio a partir de la geometría
    // definida previamente
    edificio = new Fpolygon2D(ruta);
```

```
    // agregamos el edificio al mapa
    mapa.addShape(edificio, new Object[] {
      ValueFactory.createValue(1),
      ValueFactory.createValue("Casa unifamiliar.") });

    // definimos la geometría del camino
    ruta = new GeneralPathX();
    ruta.moveTo(356242.7, 4727498.8);
    ruta.lineTo(356268.0, 4727509.8);
    ruta.lineTo(356281.0, 4727519.1);

    // creamos el objeto camino a partir de la geometría
    // definida previamente
    camino = new Fpolyline2D(ruta);

    // agregamos el camino al mapa
    mapa.addShape(camino, new Object[] {
      ValueFactory.createValue(2),
      ValueFactory.createValue("Camino de herradura.") });

    // crea la capa
    vistaActual = (View)PluginServices.getMDIManager().
      getActiveWindow();
    controlMapa = vistaActual.getMapControl();

    capa = LayerFactory.createLayer("Construcciones", mapa,
      controlMapa.getProjection());
    capa.setVisible(true);

    // agrega la capa a la vista actual
    controlMapa.getMapContext().getLayers().addLayer(capa);
  }

  public boolean isEnabled() {
    IWindow ventanaActiva;

    // obtiene la ventana activa
    ventanaActiva = PluginServices.getMDIManager().
      getActiveWindow();

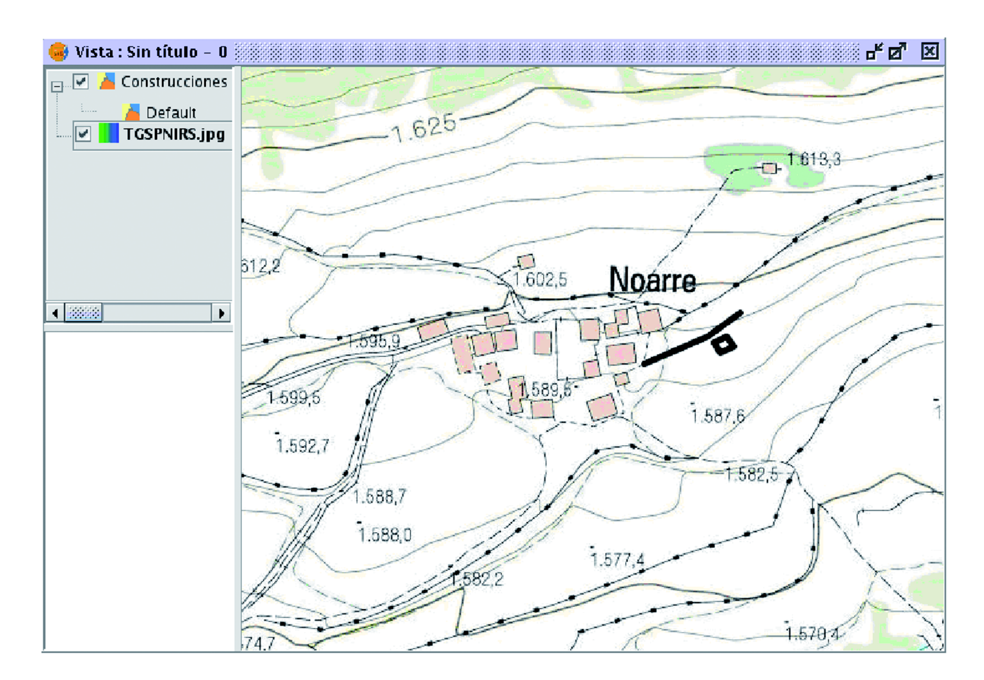
    // estará habilitada sólo si la ventana actual es una
    // ventana de vista
    return ventanaActiva instanceof View;
  }
```

```
    public boolean isVisible() {
      // siempre estará visible
      return true;
    }

    public void initialize() {

    }
  }
```

Si ejecutamos la extensión sobre una ventana de vista, el resultado será parecido al siguiente:

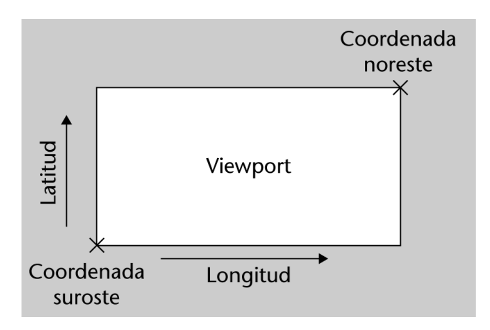


Como podemos apreciar, la vista contiene dos capas: la base topográfica ("TGSPNIRS.jpg") y la capa "Construcciones" que acabamos de crear. Mediante las clases "GPolygon2D" y "GPolyline2D" hemos conseguido dibujar un edificio y un camino al este del pueblo de Noarre.

1.7. El *viewport*

Cuando se muestra un mapa por pantalla, no siempre se muestra en toda su extensión. Que veamos una parte mayor o menor del mismo depende del nivel de *zoom*. De hecho, el área visible y el nivel de *zoom* se rigen por una regla de proporcionalidad inversa: a mayor *zoom*, menor área, y a menor *zoom*, mayor área.

El recuadro que delimita el área visible de un mapa se llama *viewport*. Normalmente, se define por las coordenadas geográficas de sus esquinas inferior izquierda (suroeste) y superior derecha (noreste). Latitud y longitud crecen desde la coordenada suroeste hasta la coordenada noreste.



gvSIG nos da acceso al *viewport* mediante la función *getViewPort* de la clase "MapControl".

El *viewport* es un objeto importante en gvSIG, pues nos proporciona mecanismos para transformar las coordenadas de la pantalla en coordenadas geográficas y viceversa. Esto nos servirá, por ejemplo, para determinar las coordenadas de un punto seleccionado mediante el ratón.

Suponiendo que las variables *x* e *y* (de tipo *int*) son las componentes de un punto del *viewport*, el código siguiente:

```
View vistaActiva;
MapControl controlMapa;
ViewPort viewport;
Point2D geoPoint;

// obtiene el viewport
vistaActiva = (View)PluginServices.getMDIManager().
  getActiveWindow();
controlMapa = vistaActiva.getMapControl();
viewport = controlMapa.getViewPort();

// transforma las coordenadas del viewport en coordenadas
// geográficas
geoPoint = viewport.toMapPoint(x, y);

JOptionPane.showMessageDialog(null, "Latitud: " +
  geoPoint().getX() + " Longitud: " +
  geoPoint.getY());
```

muestra por pantalla las coordenadas geográficas que se corresponden con este punto. Como se puede observar, la función que hace la transformación es *toMapPoint*. Esta función devuelve un objeto "Point2D" que contiene las coordenadas geográficas resultantes, cuyas componentes se pueden consultar mediante las funciones *getY* (latitud) y *getX* (longitud).

Otras funciones interesantes que nos ofrece la clase *viewport* son las siguientes:

- ***fromMapPoint***. Hace la transformación inversa: de coordenadas geográficas a coordenadas del *viewport*.

- ***fromMapDistance*.** Transforma una distancia geográfica en una distancia del *viewport*.
- ***toMapDistance*.** Transforma una distancia del *viewport* en una distancia geográfica.
- ***getAdjustedExtend*.** Devuelve los límites geográficos del *viewport*, es decir, las coordenadas de las esquinas.

1.8. Herramientas

Hasta ahora, hemos visto cómo agregar datos al mapa de manera estática, es decir, desde el mismo código del complemento, pero en ningún momento hemos abordado la interacción con el usuario. Sin embargo, es rara la aplicación SIG que no ofrezca al usuario algún tipo de mecanismo interactivo para consultar los datos relevantes de los objetos que aparecen en el mapa.

Durante el proceso de creación del mapa vectorial, hemos definido una serie de atributos para todos los elementos del mapa. Más adelante, para cada línea y polígono, hemos asignado valores concretos a estos atributos. Ahora nuestro objetivo será crear una herramienta que permita al usuario seleccionar un elemento y consultar la información que tenga asociada.

El mecanismo de acción será el siguiente:

- El usuario activará la herramienta mediante un botón de la barra de herramientas.
- El usuario seleccionará un elemento del mapa haciendo clic sobre él.
- Aparecerá una ventana en la que se mostrarán los valores de los atributos del elemento.

Veamos paso a paso cómo implementar esta funcionalidad.

1.8.1. Agregar un botón a la barra de herramientas

El primer paso será agregar un botón a la barra de herramientas de la aplicación. Los botones, como todo elemento gráfico, pueden añadirse mediante el fichero de configuración *config.xml*. Las líneas siguientes:

```
<tool-bar name="Postgrado SIG" position="2">
  <action-tool icon="seleccion.png"
    tooltip="Localizador" position="1"
    action-command="BOTON_LOCALIZADOR"/>
</tool-bar>
```

añaden un botón a la barra de herramientas de gvSIG. El significado de cada uno de los elementos se detalla a continuación:

- ***tool-bar***. Define una barra de herramientas. Este elemento tiene dos atributos:
  - ***name***: especifica el nombre de la barra de herramientas.
  - ***position***: especifica la posición que ocupará la barra de herramientas entre las demás.

- ***action-tool***. Define un botón de la barra de herramientas. Este elemento, que debe ir siempre dentro de un elemento “tool-bar”, tiene cuatro atributos:
  - ***icon***: especifica el nombre del fichero que contiene la imagen que aparecerá en el botón.
  - ***tooltip***: especifica el texto que se mostrará para describir la herramienta.
  - ***position***: especifica la posición del botón dentro de la barra de herramientas.
  - ***action-command***: al igual que el atributo “action-command” de los menús, define un identificador para la acción (este identificador se utilizará en las llamadas a la función *exec* de la extensión).

El elemento “tool-bar” debe ir siempre dentro de un elemento “extension”. Si incorporamos estas líneas al fichero de configuración de la extensión “Construcciones”, el resultado será el siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<plugin-config>
  <depends plugin-name="com.iver.cit.gvsig" />
  <libraries library-dir="."/>
  <extensions>
    <extensionclass-name=
      "edu.uoc.postgradosig.construcciones.Construcciones"
      description="Complemento que dibuja construcciones"
      active="true"
      priority="50">
      <menu text="Postgrado SIG/Construcciones"
        action-command="MENU_CONSTRUCCIONES" />
    <tool-bar name="Postgrado SIG" position="2">
      <action-tool icon="consulta.png"
        tooltip="Consultar construcciones"
        position="1"
        action-command="BOTON_CONSTRUCCIONES"/>
    </tool-bar>
    </extension>
  </extensions>
</plugin-config>
```

Es importante observar que el atributo “action-command” del menú y del botón son diferentes: en el primer caso es MENU_CONSTRUCCIONES, mientras que en el segundo caso es BOTON_CONSTRUCCIONES. Esto permite a la extensión (mediante el parámetro que recibe en la función *execute*) averiguar por medio de qué elemento de la interfaz de usuario ha sido llamada: si por medio del menú o por medio del botón de la herramienta.

El efecto sobre la interfaz de usuario de gvSIG será el siguiente:



Como se puede observar, se añade un botón “C” a la barra de herramientas.

1.8.2. Escuchar los eventos del ratón

Una vez configurada la extensión, debemos prepararla para que sea capaz de escuchar los eventos del ratón. Concretamente, nos interesa el evento “clic”, que se produce cuando se hace clic sobre un punto del mapa. Esto se hace implementando la interfaz* “PointListener” y registrando la clase como escucha de dichos eventos.

* Una interfaz es como una especie de clase cuyas funciones están definidas, pero carecen de código. Implementar una interfaz es similar a extender una clase, aunque nos veremos obligados a proporcionar una implementación para todas y cada una de las funciones que ésta defina. Generalmente, se usa para obligar a determinadas clases a implementar un conjunto de funciones que se utilizarán para comunicarse con ella. Una clase puede implementar una o más interfaces.

La interfaz “PointListener” define las siguientes funciones, que deben ser implementados obligatoriamente por la extensión:

- ***void point(PointEvent).*** Se llama a esta función cada vez que se hace clic sobre un mapa. El objeto “PointEvent” contiene las coordenadas sobre las que se ha desatado el evento.

- ***void pointDoubleClick(PointEvent).*** Esta función tiene el mismo cometido que la anterior, pero para el doble clic. El objeto “PointEvent” también contendrá las coordenadas sobre las que se ha desatado el evento.

- ***boolean cancelDrawing().*** Indica a gvSIG si debe cancelarse el dibujado en curso. Un entorno estable puede ser deseable para determinadas herramientas.

- ***Cursor getCursor().*** Devuelve el cursor de la herramienta. Mientras la herramienta está activa, el ratón mostrará este cursor.

Las líneas siguientes reflejan una posible implementación de estas funciones por parte de la extensión "Construcciones", que mostraría las coordenadas del punto seleccionado cada vez que se hiciese clic sobre el mapa. Observad que no se reproduce la clase al completo:

```
public class Construcciones extends Extension
  implements PointListener {

  (...)

  public void point(PointEvent event)
    throws BehaviourException {
    Point2D puntoViewport;

    // obtiene el punto seleccionado
    puntoViewport = event.getPoint();

    // muestra las coordenadas del punto
    JOptionPane.showMessageDialog(null,
      "x=" +puntoViewport.getX() +
      "y=" +puntoViewport.getY());
  }

  public void pointDoubleClick(PointEvent event)
    throws BehaviourException {
    // ante un doble clic no hace nada
  }

  public boolean cancelDrawing() {
    return false; // no nos interesa cancelar el dibujado
  }

  public Cursor getCursor() {
    return null; // no queremos cambiar el cursor
  }
}
```

Como podemos ver, la función *point* obtiene el punto seleccionado por medio de la función *getPoint* del objeto "event" que recibe por parámetro. Observad el uso de la palabra clave *implements* en la declaración de la clase, seguida del nombre de la clase (o clases) que implementa (en este caso, "PointListener"). Observad también el uso de la fórmula *throws BehaviourException* en la declaración de la función *point*, que permite al código indicar un estado de excepción*.

*** Las excepciones son un mecanismo muy potente de control de errores. Sin embargo, su estudio va más allá de los objetivos de esta asignatura.**

Implementar la interfaz "PointListener" capacita a la clase para recibir los clics del ratón, pero esto no significa que los reciba de inmediato: es necesario de-

clarar la voluntad de recibir dichos eventos. Esto se lleva a cabo mediante la función *addMapTool* de la clase “MapControl”:

```
controlMapa.addMapTool("Construcciones",
  new PointBehavior(this));
```

En el ejemplo, “controlMapa” es un objeto de la clase “MapControl”. Lo que hacemos realmente es registrar nuestra clase como una herramienta de la ventana de vista que actúa en respuesta a los clics del ratón. Como se puede observar, la función *addMapTool* espera dos parámetros, el nombre de la herramienta y un objeto “PointBehavior”, que es el que representa el comportamiento de la extensión. De este segundo parámetro, que no variará de una extensión a otra, sólo hay que destacar el uso de la palabra *this* para referirse al objeto actual.

Para activar la herramienta, utilizaremos la sentencia siguiente:

```
controlMapa.setTool(“Construcciones”);
```

Una vez hecho esto, cualquier evento relacionado con un punto del mapa será notificado a nuestra extensión.

1.8.3. Obtener un elemento del mapa

Obtenidas las coordenadas de un punto, nos interesa averiguar qué elemento del mapa se encuentra en el punto seleccionado. Normalmente encontraremos alguno de los elementos que hemos incorporado al mapa (líneas y polígonos), aunque es posible que no hallemos ninguno si el usuario ha hecho clic sobre un área vacía.

La consulta se realiza mediante la función *queryByPoint* de la clase “FLyrVect”:

```
elementos = (FLyrVect)capa.queryByPoint(puntoGeo,
  mapControl.getViewPort().toMapDistance(1));
```

donde “puntoGeo” es un objeto de la clase “Point2D” y “capa” uno de la clase “FLyrVect”, que representan el punto y la capa que se desea consultar respectivamente. La función devuelve un objeto “FBitSet” que contiene los datos asociados al elemento encontrado.

La función *queryByPoint* espera un segundo parámetro que es el radio de la consulta. En nuestro caso, consideramos suficiente la distancia equivalente a un píxel* del *viewport*. Como la función espera una distancia real, usamos la

*** Un píxel es la unidad mínima de representación de un dispositivo gráfico. Corresponde a un punto de color.**

función *toMapDistance* del *viewport* para trasladar la distancia de un píxel a su equivalente en la realidad.

Los datos obtenidos los recuperaremos mediante la función *getFieldValue* del objeto devuelto. Para determinar si hay datos y, en consecuencia, si la consulta ha encontrado algún elemento en la posición indicada, utilizaremos la función *isEmpty*.

Dado un objeto "capa" de la clase "FLyrVect" y un objeto "elementos" de la clase "FBitSet", el código siguiente:

```
DataSource fuenteDatos;
Value id, descripcion;

if (!elementos.isEmpty()) { // si hay datos
  // obtiene el objeto que contiene los datos
  fuenteDatos = ((AlphanumericData)capa).getRecordset();

  // empieza la lectura de los datos
  fuenteDatos.start();

  // lee el atributo "ID"
  id = fuenteDatos.getFieldValue(elementos.nextSetBit(0),
    fuenteDatos.getFieldIndexByName("ID"));

  // lee el atributo "Descripcion"
  descripcion = fuenteDatos.getFieldValue(
    elementos.nextSetBit(0),
    fuenteDatos.getFieldIndexByName("Descripcion"));

  // finaliza la lectura de los datos
  fuenteDatos.stop();
}
```

obtiene los valores de los campos "ID" y "Descripcion" del elemento encontrado y los almacena en las variables "id" y "descripcion" respectivamente. Observad que la función *getFieldValue* espera dos parámetros: la fila y la columna del elemento que se desea leer. Debemos imaginarnos "fuenteDatos" como una tabla en la que las filas son los elementos del mapa y las columnas, sus atributos. La fila la obtenemos del objeto "elementos" mediante la función *nextSetBit*, que nos devuelve la fila del primer objeto encontrado. Por su parte, la columna la obtenemos mediante la función *getFieldIndexByName*, que nos devuelve el número de columna de un atributo a partir de su nombre*.

*** El número de columna de los atributos depende del orden en el que se hayan definido. En el ejemplo, se definen los atributos "ID" y "Descripcion" por este orden. En consecuencia, el atributo "ID" estará en la columna 0, mientras que el atributo "Descripcion" estará en la columna 1.**

Juntándolo todo, el código de la extensión nos queda así:

```
package edu.uoc.postgradosig.construcciones;

import java.awt.Cursor;
import java.awt.geom.Point2D;

import javax.swing.JOptionPane;

import com.hardcode.gdbms.engine.data.DataSource;
import com.hardcode.gdbms.engine.values.Value;
import com.hardcode.gdbms.engine.values.ValueFactory;
import com.iver.andami.PluginServices;
import com.iver.andami.plugins.Extension;
import com.iver.andami.ui.mdiManager.IWindow;
import com.iver.cit.gvsig.fmap.MapControl;
import com.iver.cit.gvsig.fmap.ViewPort;
import com.iver.cit.gvsig.fmap.core.FPolygon2D;
import com.iver.cit.gvsig.fmap.core.FPolyline2D;
import com.iver.cit.gvsig.fmap.core.FShape;
import com.iver.cit.gvsig.fmap.core.GeneralPathX;
import com.iver.cit.gvsig.fmap.drivers.ConcreteMemoryDriver;
import com.iver.cit.gvsig.fmap.layers.FBitSet;
import com.iver.cit.gvsig.fmap.layers.FLayer;
import com.iver.cit.gvsig.fmap.layers.FLyrVect;
import com.iver.cit.gvsig.fmap.layers.LayerFactory;
import com.iver.cit.gvsig.fmap.layers.layerOperations.
  AlphanumericData;
import com.iver.cit.gvsig.fmap.tools.BehaviorException;
import com.iver.cit.gvsig.fmap.tools.Behavior.PointBehavior;
import com.iver.cit.gvsig.fmap.tools.Events.PointEvent;
import com.iver.cit.gvsig.fmap.tools.Listeners.
  PointListener;
import com.iver.cit.gvsig.project.documents.view.gui.View;

public class Construcciones extends Extension
  implements PointListener {
  private FLayer capa;

  public void accionMenu() {
    ConcreteMemoryDriver mapa;
    GeneralPathX ruta;
    FShape edificio, camino;
    MapControl controlMapa;
    View vistaActual;
```

```
    // crea un mapa vectorial en memoria
    mapa = new ConcreteMemoryDriver();
    mapa.setShapeType(FShape.LINE + FShape.POLYGON);

    // define los atributos de los elementos
    mapa.getTableModel().setColumnIdentifiers(new String[] {
      "ID", "Descripcion"});

    // agrega el edificio al mapa
    ruta = new GeneralPathX();
    ruta.moveTo(356270.7, 4727507.1);
    ruta.lineTo(356273.0, 4727503.3);
    ruta.lineTo(356278.5, 4727505.8);
    ruta.lineTo(356275.5, 4727509.8);
    ruta.lineTo(356270.7, 4727507.1);

    edificio = new FPolygon2D(ruta);
    mapa.addShape(edificio, new Object[] {
      ValueFactory.createValue(1),
      ValueFactory.createValue("Casa unifamiliar.") });

    // agrega el camino al mapa
    ruta = new GeneralPathX();
    ruta.moveTo(356242.7, 4727498.8);
    ruta.lineTo(356268.0, 4727509.8);
    ruta.lineTo(356281.0, 4727519.1);

    camino = new FPolyline2D(ruta);
    mapa.addShape(camino, new Object[] {
      ValueFactory.createValue(2),
      ValueFactory.createValue("Camino de herradura.") });

    // crea la capa...
    vistaActual = (View)PluginServices.getMDIManager().
      getActiveWindow();
    controlMapa = vistaActual.getMapControl();

    capa = LayerFactory.createLayer("Construcciones", mapa,
      controlMapa.getProjection());
    capa.setVisible(true);

    // ... y la agrega a la vista actual
    controlMapa.getMapContext().getLayers().addLayer(capa);
  }

  public void accionBoton() {
    View vistaActual;
    MapControl controlMapa;
```

```
    // obtiene la vista actual
    vistaActual = (View) PluginServices.getMDIManager().
      getActiveWindow();
    controlMapa = vistaActual.getMapControl();

    // registra la escucha
    controlMapa.addMapTool("Construcciones",
      new PointBehavior(this));
    controlMapa.setTool("Construcciones");
  }

  public void execute(String actionCommand) {
    // realiza una accion u otra dependiendo de la opción
    // de menú seleccionada
    if (actionCommand.equals("MENU_CONSTRUCCIONES")) {
      accionMenu();
    }
    else if (actionCommand.equals("BOTON_CONSTRUCCIONES")) {
      accionBoton();
    }
  }

  public boolean isEnabled() {
    IWindow ventanaActiva;

    ventanaActiva = PluginServices.getMDIManager().
      getActiveWindow();

    // devuelve true en caso de que ventanaActiva sea de la
    // clase View
    return ventanaActiva instanceof View;
  }

  public void initialize() {

  }

  public boolean isVisible() {
    return true;
  }

  public void point(PointEvent event)
    throws BehaviorException {
    View vistaActiva;
    Point2D puntoGeo;
    ViewPort viewport;
    FBitSet elementos;
    MapControl controlMapa;
```

```
  DataSource fuenteDatos;
  Value id, descripcion;

  // obtiene el viewport
  vistaActiva = (View)PluginServices.getMDIManager().
    getActiveWindow();
  controlMapa = vistaActiva.getMapControl();
  viewport = controlMapa.getViewPort();

  // obtiene el punto en el que se ha producido el clic
  puntoGeo = viewport.toMapPoint(event.getPoint());
  try {
    elementos = ((FLyrVect) capa).queryByPoint(puntoGeo,
      viewport.toMapDistance(1));

    if (!elementos.isEmpty()) { // si hay datos
      // obtiene el objeto que contiene los datos
      fuenteDatos = ((AlphanumericData)capa).
        getRecordset();

      // empieza la lectura de los datos
      fuenteDatos.start();

      // lee el atributo "ID"
      id = fuenteDatos.getFieldValue(
        elementos.nextSetBit(0),
        fuenteDatos.getFieldIndexByName("ID"));

      // lee el atributo "Descripcion"
      descripcion = fuenteDatos.getFieldValue(
        elementos.nextSetBit(0),
        fuenteDatos.getFieldIndexByName("Descripcion"));

      // finaliza la lectura de los datos
      fuenteDatos.stop();

      // muestra los datos
      JOptionPane.showMessageDialog(null, "ID = " + id +
        "; Descripcion = \"" + descripcion + "\"");
    }
  }
  catch (Exception e) {

  }
}

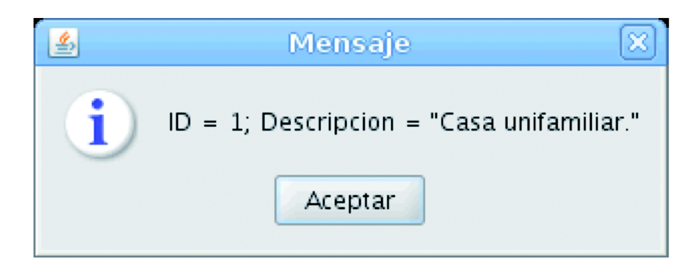
public void pointDoubleClick(PointEvent arg0)
  throws BehaviorException {
  // no hace nada
}
```

```
  public boolean cancelDrawing() {
    return false; // no nos interesa cancelar el dibujado
  }

  public Cursor getCursor() {
    return null; // sin cursor
  }
}
```

Hemos de destacar la presencia de un bloque *try* – *catch* en la función *point*. Este bloque es necesario para gestionar las excepciones que se pueden producir durante la llamada a las funciones *getRecorset*, *start*, *getFieldValue* y *stop*.

Si se selecciona la herramienta y se hace clic sobre el edificio que hemos creado, obtendremos el resultado siguiente:



Resumen

En este módulo hemos aprendido a extender una herramienta SIG para adaptarla a nuestras necesidades. Hemos aprendido a crear una capa con información vectorial y a rellenarla con elementos georreferenciados. Además, hemos visto cómo asociar datos a los elementos del mapa y cómo recuperarlos más tarde por medio de un simple clic del ratón.

Aunque en los ejemplos hemos trabajado exclusivamente con datos vectoriales almacenados en el código de la extensión, con unos pocos cambios estos se pueden leer de un fichero o de cualquier otra fuente de datos externa. Del mismo modo, las posibilidades de extensión de gvSIG van más allá de la representación estática de datos vectoriales, pudiendo representar datos a través de iconos o elementos en movimiento. En este sentido, las posibilidades que nos ofrece un lenguaje como Java, combinadas con la flexibilidad de gvSIG, son casi infinitas.

Bibliografía

Victor Acevedo, "*Guía de referencia para gvSIG 1.1*", http://www.gvsig.org/web/docdev/reference/.

Francisco José Peñarrubia, "*Manual para desarrolladores gvSIG v1.1*", http://www.gvsig.org/web/docdev/manual-para-desarrolladores-gvsig/.

Asociación gvSIG, Código fuente (Javadoc) de la versión 1.10 de gvSIG, http://www.gvsig.org/web/projects/gvsig-desktop/official/gvsig-1.10/.

Andres Herrera, "*Escribir un PlugIng para gvSIG "from scratch" en M$-Windows*", http://t763rm3n.googlepages.com/PlugIng_gvSIG-fromScratch_Borrador.pdf.

Jorge Piera, "*Crear una extensión desde 0 en gvSIG*", http://forge.osor.eu/docman/view.php/89/362/Crear_extensiones_en_gvSIG.pdf.

Programación SIG en entornos web

Albert Gavarró Rodríguez

PID_00174757



Índice

Introducción

En el módulo anterior, hemos aprendido a extender un SIG de escritorio (gvSIG) para adaptarlo a nuestras necesidades. También hemos aprendido a agregar nuevas capas por código y elementos georeferenciados en ellas.

En este bloque aprenderemos a hacer prácticamente lo mismo pero en entornos web. Estudiaremos cómo construir una página web y cómo incorporarle un mapa. Aprenderemos también a agregar elementos georeferenciados al mapa que nos servirán para marcar puntos, trazar caminos o delimitar áreas.

Aunque hay varios servicios de mapas que operan en Internet, como Yahoo Maps o Microsoft Virtual Earth, este módulo se centrará en el estudio de Google Maps. Sin embargo, las técnicas que se explicarán serán fácilmente trasladables a los otros servicios.

Objetivos

Al final de este módulo, deberíamos ser capaces de:

1. Crear páginas web simples en las que se muestre un mapa.

2. Mostrar un tipo de mapa u otro dependiendo de nuestras necesidades.

3. Agregar marcadores georreferenciados a un mapa para señalar lugares.

4. Dibujar líneas y polígonos georreferenciados en un mapa para mostrar rutas o delimitar áreas.

5. Convertir las coordenadas de un mapa a coordenadas geográficas y viceversa.

1. El lenguaje HTML

1.1. La Web y el HTML

La World Wide Web, que se abrevia WWW y se conoce también como la Web, es un sistema de páginas enlazadas mediante el concepto de hipertexto. El hipertexto es un texto que refiere a otro texto al que el lector puede tener acceso inmediato. El lenguaje tradicional y predominante sobre el que se sustenta la Web y que tiene capacidades de hipertexto es el HTML (*HyperText Markup Language*, 'lenguaje de marcas de hipertexto').

Hasta hace relativamente poco era necesario conocer el lenguaje HTML para poder publicar una página web en Internet. Si bien el HTML es un lenguaje muy directo, en el que buena parte del código es el contenido propiamente dicho del documento, como contrapartida es engorroso de escribir. Así, por ejemplo, para mostrar una palabra en negrita, es necesario intercalar en el texto más de media docena de caracteres extras.

Hoy en día han aparecido muchísimas herramientas que permiten generar páginas web de la misma forma que editamos cualquier otro documento. De hecho, casi todos los procesadores de textos modernos ofrecen la posibilidad de guardar sus documentos en formato HTML. Además, hay herramientas altamente especializadas que generan páginas de altísima calidad en poco tiempo y con el mínimo esfuerzo, como por ejemplo Amaya*, Bluefish** o Dreamweaver***.

*** http://www.w3.org/Amaya**

**** http://bluefish.openoffice.nl/**

***** http://www.adobe.com/products/dreamweaver**

Dada la existencia de estas herramientas de edición y puesto que el objetivo de estos materiales no es profundizar en el conocimiento del HTML, sólo daremos unas pocas pinceladas sobre este lenguaje, las necesarias para escribir páginas relativamente sencillas y para comprender los ejemplos que se trabajarán en el resto del módulo. Sin embargo, la UOC dispone de materiales abiertos que profundizan en el lenguaje HTML*.

*** http://mosaic.uoc.edu/2009/11/20/introduccion-a-la-creacion-de-paginas-web/**

1.2. Elementos y etiquetas

El lenguaje HTML se utiliza para describir la estructura y el contenido de un documento, así como para complementar el texto con objetos (por ejemplo, mapas) e imágenes.

Un típico código HTML está formado por elementos que configuran las distintas partes del documento. Generalmente, estos elementos están delimitados por una etiqueta de inicio y otra de fin, que se escriben siempre entre corchetes angulares ("<" y ">").

Por ejemplo, la línea siguiente:

```
<p>La WWW nació en el CERN.</p>
```

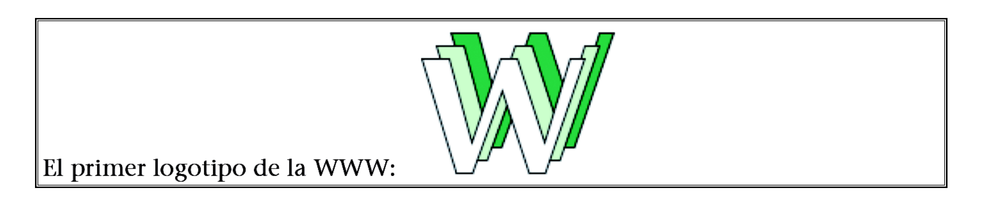
define un elemento (párrafo) cuyo contenido es “La WWW nació en el CERN.”. Como podemos ver, las etiquetas de inicio y fin (<p> y </p> respectivamente) delimitan el contenido del párrafo. Además, la etiqueta de fin es igual que la de inicio pero precedida por una barra (“/”).

Hay elementos que sólo tienen una etiqueta de inicio, pues carecen de contenido textual. Es el caso, por ejemplo, de las imágenes. Las líneas siguientes*:

*** El símbolo ↵ indica la ruptura de una línea de texto por motivos de formato. En consecuencia, las líneas separadas por este símbolo deberían considerarse como una de sola.**

```
El primer logotipo de la WWW:
<img src="http://upload.wikimedia.org/wikipedia/ ↵
commons/thumb/b/b2/WWW_logo_by_Robert_Cailliau.svg/ ↵
120px-WWW_logo_by_Robert_Cailliau.svg.png">
```

muestran la imagen que se encuentra en la dirección:
http://upload.wikimedia.org/wikipedia/commons/thumb/b/b2/WWW_logo_by_Robert_Cailliau.svg/120px-WWW_logo_by_Robert_Cailliau.svg.png. El resultado será el siguiente:



Observad la estructura que aparece dentro de la etiqueta de inicio:

```
src="http://upload.wikimedia.org/wikipedia/commons ↵
thumb/b/b2/WWW_logo_by_Robert_Cailliau.svg/120px ↵
WWW_logo_by_Robert_Cailliau.svg.png"
```

Todos los elementos HTML tienen atributos que varían de un elemento a otro según sus características. Dichos atributos permiten variar la forma en la que se presenta el elemento. En el caso de las imágenes, tienen un atributo “src” (de *source*, ‘fuente’) que indica la dirección web en la que se encuentra la imagen. Como se puede apreciar en el ejemplo, el valor de los atributos siempre especifíca en la etiqueta de inicio.

Una última característica importante de los elementos HTML es que se pueden imbricar. Es decir, en su contenido un elemento puede contener otros elementos que, a su vez, pueden contener otros, y así sucesivamente, formando una jerarquía de elementos.

El código siguiente:

```
<p>WWW significa <em>World Wide Web</em>.</p>
```

define un párrafo que contiene una texto enfatizado (*World Wide Web*). Es importante observar que los elementos se cierran en orden inverso a aquel en el que se han abierto. El resultado será el siguiente:

WWW significa *World Wide Web*.

1.3. Estructura mínima de un documento HTML

Un documento HTML bien formado debe presentar siempre la siguiente estructura mínima:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="es">
<head>
<title>Título del documento</title>
</head>
<body>
Contenido del documento.
</body>
</html>
```

Como podemos observar, la estructura mínima de un documento HTML está formada por cinco elementos:

- **DOCTYPE.** Especifica la versión del lenguaje HTML que obedece el documento (en el ejemplo, la 4.01).

- **HTML.** Es el elemento raíz, el que define el documento HTML en sí. Contiene obligatoriamente los elementos *head* y *body*. Como puede apreciarse en el ejemplo, el elemento *html* puede contener el atributo *lang*, que define el idioma del documento (“es” de español).

- **HEAD.** Define la cabecera del documento, cuyo cometido es alojar información sobre el mismo. El título, las palabras clave y otros datos que no se consideran parte del contenido del documento irán dentro de este elemento.

- **TITLE.** Define el título del documento.

- **BODY.** Define el contenido del documento. Este elemento contendrá otros que definirán el contenido del documento.

1.4. Definición del contenido

El contenido de un documento HTML puede definirse mediante tres tipos de elementos:

- **Estructurales.** Son aquellos que describen la estructura del texto. En HTML tenemos elementos estructurales para definir títulos, párrafos, tablas, enumeraciones, etc. Normalmente, los navegadores aplican diferentes estilos de presentación a cada elemento.

- **De presentación.** Son aquellos que describen el aspecto del texto, independientemente de su función. Los elementos que definen negritas, cursivas, texto enfatizado, etc. son elementos de presentación.

- **De hipertexto.** Son aquellos que permiten enlazar con otros documentos mediante la creación de hipervínculos. Son los más importantes dada la naturaleza hipertextual de la Web.

A continuación, se mostrarán algunos de los elementos más significativos de cada tipo. Además, ilustraremos su uso con ejemplos.

1.4.1. Elementos estructurales

Como ya se ha mencionado, los elementos estructurales son aquellos que describen el propósito del texto. Entre ellos podemos distinguir los siguientes:

- ***P,*** de *paragraph* (párrafo). Define un párrafo de texto.

Las líneas siguientes:

```
<p>La WWW es un sistema de páginas enlazadas mediante el
concepto de hipertexto.</p>
<p>La WWW nació en 1989 en el CERN, de la mano de Tim
Berners-Lee.</p>
```

definen un par de párrafos. El resultado será el siguiente:

> La WWW es un sistema de páginas enlazadas mediante el concepto de hipertexto.
>
> La WWW nació en 1989 en el CERN, de la mano de Tim Berners-Lee.

Como podemos observar, al final de cada párrafo se agrega automáticamente un salto de línea.

- ***H1 ... H6,*** de *heading* (título). Definen hasta seis niveles de títulos (desde el más importante —*h1*— hasta el menos importante —*h6*. Por ejemplo, las líneas siguientes:

```
<h1>Introducción</h1>
<p>La WWW es un sistema de páginas enlazadas mediante el
concepto de hipertexto.</p>
<h2>Historia</h2>
<p>La WWW nació en 1989 en el CERN, de la mano de Tim
Berners-Lee.</p>
```

definen un título de primer nivel (“Introducción”) y otro de segundo nivel (“La Web y el HTML”). Después de cada título, aparece algo de texto en forma de párrafo. El resultado será el siguiente:

> **Introducción**
>
> La WWW es un sistema de páginas enlazadas mediante el concepto de hipertexto.
>
> **La Web y el HTML**
>
> La WWW nació en 1989 en el CERN, de la mano de Tim Berners-Lee.

- ***DIV,*** de *divider* (divisor). Este elemento define un bloque que puede contener otros elementos. Es muy parecido al párrafo, aunque está vacío de significado. El elemento *div* es muy importante porque nos permite agrupar elementos sin caracterizarlos. Las líneas siguientes:

```
<div align="center">
<p><img src="http://upload.wikimedia.org/wikipedia/
commons/thumb/b/b2/WWW_logo_by_Robert_Cailliau.svg/
120px-WWW_logo_by_Robert_Cailliau.svg.png"></p>
<p>El primer logotipo del WWW</p>
</div>
```

crean un bloque que contiene dos párrafos y lo centran en medio del documento mediante el atributo “align” (de *alignment* —alineación—). Consecuentemente, los párrafos también quedarán alineados en el centro. El resultado será el siguiente:



El primer logotipo de la WWW

- ***BR,*** de *line break* (salto de línea). Permite emplazar un salto de línea, por ejemplo, en medio de un párrafo. Este elemento no tiene etiqueta de fin. El código siguiente:

```
<p>En la página <br>http://www.w3c.org<br> se puede
obtener información sobre el WWW Consortium.</p>
```

escribirá:

En la página:
http://www.w3c.org
se puede obtener información sobre el WWW Consortium.

Observad que, a diferencia de lo que sucede en el párrafo, el salto de línea no modifica el espaciado entre líneas.

1.4.2. Elementos de presentación

Como ya hemos mencionado, los elementos de presentación son aquellos que describen el aspecto del texto, independientemente de su función. Entre ellos destacan los siguientes:

- ***EM***, de *emphasis* (énfasis). Este elemento enfatiza el texto. Normalmente, el texto se mostrará en cursiva. Las líneas siguientes:

```
<p>La <em>World Wide Web</em> es un sistema de páginas
enlazadas mediante el concepto de hipertexto.</p>
```

se traducirán en:

La *World Wide Web* es un sistema de páginas enlazadas mediante el concepto de hipertexto.

- ***STRONG***, de *strong emphasis* (énfasis fuerte). Este elemento es parecido al anterior, pero pone un énfasis aún mayor. Normalmente, el texto se mostrará en negrita. Las líneas siguientes:

```
<p>La <strong>World Wide Web</strong> es un sistema de
páginas enlazadas mediante el concepto de hipertexto.</p>
```

se traducirán en:

La **World Wide Web** es un sistema de páginas enlazadas mediante el concepto de hipertexto.

1.4.3. Elementos de hipertexto

El elemento principal que permite navegar de un documento a otro es ***A***, de *anchor* (ancla). Dicho elemento permite definir un texto que, cuando se activa, nos lleva a otro documento. Por ejemplo, las líneas siguientes:

```
<p>La WWW nació en 1989 en el CERN, de la mano de
<a href="http://es.wikipedia.org/wiki/
Tim_Berners-Lee">Tim Berners-Lee</a>.</p>
```

crean un enlace entre el nombre del inventor de la WWW y el artículo correspondiente de la Wikipedia. Como se puede observar, el elemento *ancla* tiene un atributo *href* (de *hypertext reference*, 'referencia de hipertexto'), que establece el documento con el que está enlazado el texto. El resultado será el siguiente:

La WWW nació en 1989 en el CERN, de la mano de Tim Berners-Lee.

Al activar el enlace, ya sea haciendo clic sobre él o por otros medios, iremos a la página http://es.wikipedia.org/wiki/Tim_Berners-Lee.

2. Google Maps

Google Maps es la herramienta SIG en línea de Google. Permite explorar mapas en 2D de casi cualquier parte del mundo. La popularidad de la que goza Google Maps en la actualidad se debe principalmente a su buen rendimiento, su vasta cartografía y la posibilidad de incorporar toda esta tecnología a cualquier sitio web mediante el uso de la API* (de *Application Programming Interface*, 'interfaz de programa de aplicación') de Google Maps.

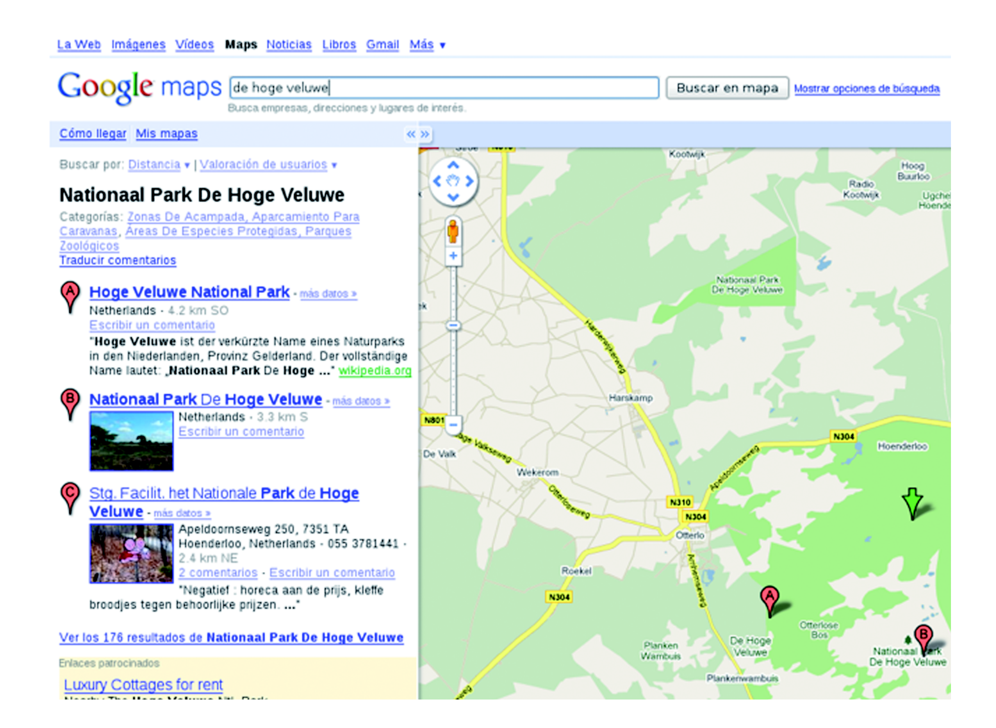
*** La documentación completa de la API se puede encontrar en:**
http://code.google.com/intl/ca/apis/maps/documentation/reference.html

En los apartados siguientes, veremos cómo incorporar un mapa a nuestra página y cómo hacer uso de la API para trabajar sobre él.

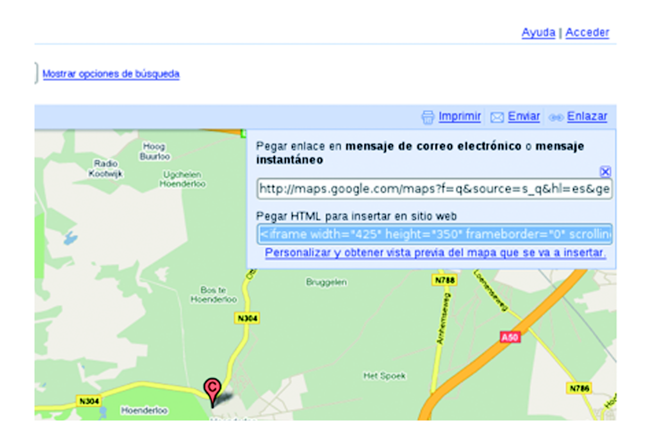
2.1. Incorporar un mapa en una página

La forma más sencilla de incorporar un mapa en nuestra página es utilizando el web de Google Maps y, una vez seleccionada la ubicación que deseamos mostrar, incorporar el código HTML que nos proporciona Google a nuestra página.

Veámoslo paso a paso. En primer lugar, debemos seleccionar una ubicación en maps.google.es. Esto se puede hacer mediante la caja de búsqueda o navegando por el mapa.



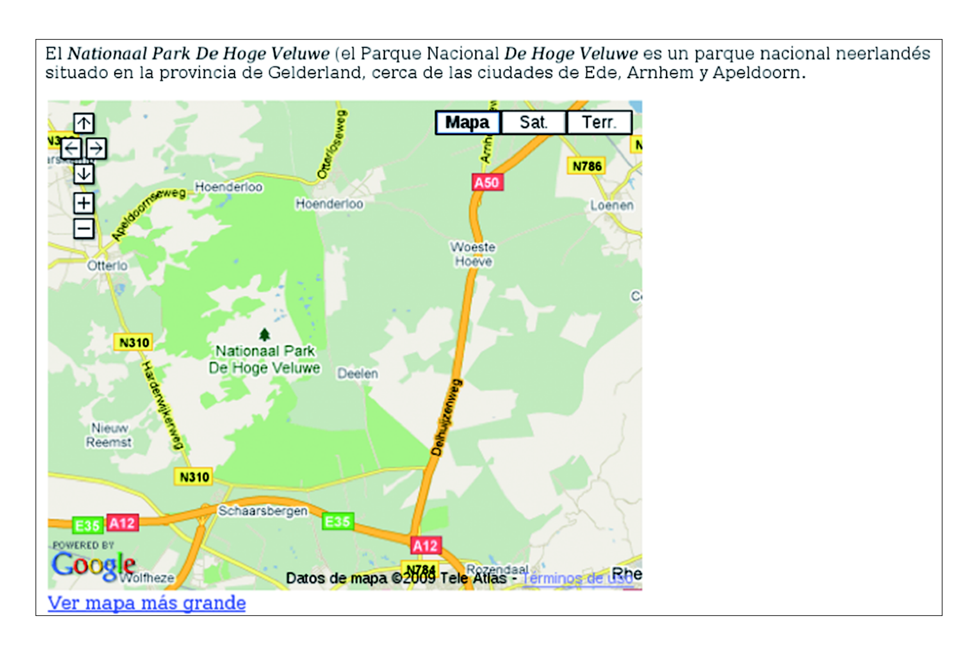
En segundo lugar, hemos de seleccionar la opción *Enlazar* y copiar el código HTML que aparece en la caja "Pegar HTML para insertar en sitio web".



El tercer y último paso será incorporar este código en nuestra página HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Nationaal Park De Hoge Veluwe</title>
</head>
<body>
<p>El <em>Nationaal Park De Hoge Veluwe</em> (el Parque
Nacional <em>De Hoge Veluwe</em> es un parque nacional
neerlandés situado en la provincia de Gelderland, cerca
de las ciudades de Ede, Arnhem y Apeldoorn.</p>
<iframe width="425" height="350" frameborder="0"
scrolling="no" marginheight="0" marginwidth="0"
src="http://maps.google.com/?hl=es&amp;ie=UTF8 ↵
&amp;ll=52.075286,5.889359&amp;spn=0.261235,0.698318 ↵
&amp;z=11&amp;output=embed">
</iframe><br />
<small><a href="http://maps.google.com/?hl=es ↵
&amp;ie=UTF8&amp;ll=52.075286,5.889359 ↵
&amp;spn=0.261235,0.698318&amp;z=11&amp;source=embed"
style="color:#0000FF;text-align:left">Ver mapa más
grande</a></small>
</body>
</html>
```

El resultado será el siguiente:



Si bien el mapa que obtenemos es interactivo (podemos hacer *zoom* y desplazarnos por él) y algo personalizable (podemos cambiar mínimamente el aspecto de nuestro mapa mediante la opción *Personalizar y obtener vista previa del mapa que se va a insertar*), las posibilidades de interacción que nos ofrece son muy limitadas. Por ejemplo, no se pueden añadir o quitar capas, y no se puede asociar información a determinados puntos del mapa. Sin embargo, si nuestra única intención es situar un lugar en el mapa, ésta es una solución fácil y rápida.

2.2. La API de Google Maps

Afortunadamente, Google Maps nos ofrece muchas más posibilidades que las que hemos visto hasta ahora. Sin embargo, éstas se obtienen a costa de la facilidad de uso, pues se requieren unos mínimos conocimientos de programación para poder sacarles partido.

A esta funcionalidad adicional se accede por medio de la API de Google Maps, mediante un conjunto de tecnologías agrupadas bajo la denominación AJAX (*Asynchronous JAvascript and XML*, 'JavaScript asíncrono y XML'). Como su nombre indica, AJAX está compuesto por dos tecnologías:

- **JavaScript.** Es un lenguaje muy parecido a Java que se usa para alterar documentos HTML de forma dinámica. Es importante remarcar que no es Java; sólo está basado en él.
- **XML.** Es un lenguaje usado para el intercambio de datos (sigla de *eXtensible Markup Language*, 'lenguaje de marcas extensible'). Es una forma genérica de escribir documentos maximizando su simplicidad, su generalidad y su usabilidad.

Ambas tecnologías, integradas en un documento HTML, permiten obtener mapas y datos de los servidores de Google Maps y mostrarlos en la página.

2.2.1. Crear un mapa

La función más elemental que nos ofrece la API de Google Maps es crear un mapa. El mapa se mostrará de forma muy parecida a cuando se copia el código HTML del sitio de Google Maps: centrado en un punto y con un nivel de *zoom* determinado. Además, se mostrarán una serie de controles mínimos que nos permitirán desplazarnos por el mapa, variar su ampliación y seleccionar el tipo de mapa mostrado.

El código siguiente muestra un mapa centrado en el rectorado de la UOC:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
content="text/html; charset=utf-8"/>
<title>Universitat Oberta de Catalunya</title>

<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>

<script type="text/javascript">
function InicializarMapa() {
  var mapa;
  var centro;

  if (GBrowserIsCompatible()) {
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    centro = new GLatLng(41.414829, 2.133003);
    mapa.setCenter(centro, 17);
    mapa.setUIToDefault();
  }
}
</script>

</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
</body>
</html>
```

Como podemos observar, el código es una mezcla entre HTML y JavaScript. El código JavaScript (que hemos resaltado convenientemente mediante recua-

dros) se incluye mediante el elemento *script* y siempre va situado en la cabecera del documento (*head*). Observad el uso del atributo “type” (‘tipo’) del elemento *script*, que identifica el código como JavaScript.

También es importante observar las similitudes entre Java y JavaScript. Quizá la diferencia más notable sea la ausencia de tipos. Todas las variables son del mismo tipo, aunque como ya veremos, internamente puedan tener atributos distintos. Consecuentemente, las variables se declaran usando la fórmula ***var*** *nombre_variable* y las funciones, ***function*** *nombre_función*.

Veamos los pasos que hay que seguir para mostrar un mapa en nuestra página: en primer lugar, es necesario crear un elemento HTML que albergue el mapa. Aunque en la práctica hay diversos elementos que pueden desempeñar esta función, es recomendable utilizar *div*, porque es el elemento más neutro. Al elemento utilizado debemos darle un nombre mediante el atributo “id” (de *identifier*, ‘identificador’) y, adicionalmente, un tamaño, que especificaremos mediante los atributos estilísticos *width* (ancho) y *height* (alto), y que será el tamaño del mapa medido en píxeles. En el ejemplo, se ha creado el elemento mediante el código siguiente:

```
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
```

Como se puede observar, al elemento se le ha llamado “elemento_mapa”, y se le han atribuido las dimensiones iniciales de 500 por 300 píxeles.

Una vez creado el elemento que albergará el mapa, debemos escribir el código JavaScript que se encargará de inicializarlo y mostrarlo. Para llevar a cabo esta tarea, hemos de realizar tres acciones:

- incluir el código de la API de Google Maps,
- escribir una función que se encargue de inicializar el mapa, y
- llamar a la función de inicialización cuando el navegador termine de dibujar la página.

Veámoslas una a una:

Incluir el código de la API de Google Maps

Mediante la primera acción, damos acceso a nuestro código a la API de Google Maps. Esta acción se realiza en el primer bloque de código JavaScript:

```
<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>
```

Por regla general, este código siempre será el mismo. Su única peculiaridad es el valor "key" (que en el ejemplo ha sido reemplazado por varias equis), que variará de un lugar web a otro y que se debe obtener de Google mediante un proceso de registro gratuito*. Una vez hecho esto, el código JavaScript de nuestra página podrá acceder a las funciones de la API de Google Maps.

* http://code.google.com/intl/ca/apis/maps/signup.html
No es necesario obtener una clave para las páginas almacenadas localmente en nuestro ordenador.

Escribir una función que se encargue de inicializar el mapa

La segunda acción crea el mapa sobre el elemento HTML seleccionado y lo inicializa, estableciendo el centro y el nivel de *zoom* iniciales. En el ejemplo, esta acción la lleva a cabo la función *InicializarMapa* en el segundo bloque de código JavaScript:

```
<script type="text/javascript">
function InicializarMapa() {
 var mapa;
 var centro;

 if (GBrowserIsCompatible()) {
  mapa = new GMap2(
   document.getElementById("elemento_mapa"));
  centro = new GLatLng(41.414829, 2.133003);
  mapa.setCenter(centro, 17);
  mapa.setUIToDefault();
 }
}
</script>
```

Como se puede observar, en primer lugar instanciamos un objeto "GMap2", que es el mapa en sí, ubicándolo en el elemento HTML "elemento_mapa", que es el que hemos creado al principio con este objetivo. Después, establecemos el centro y el nivel de *zoom* iniciales del mapa, mediante la función *setCenter* de la clase "GMap2". Como primer argumento, esta función requiere un objeto "GLatLng", que no es más que una coordenada geográfica expresada en función de su latitud y su longitud (según el sistema de coordenadas WGS84). El segundo elemento es un entero, que puede variar entre 0 y 19, por los niveles de *zoom* mínimo y máximo respectivamente.

La llamada a la función *GBrowserIsCompatible* evita la ejecución del código en caso de que se detecte que el navegador utilizado no es compatible con Google Maps.

Llamar a la función de inicialización cuando el navegador termine de dibujar la página

Finalmente, sólo queda indicar al navegador cuándo se debe inicializar el mapa o, lo que es lo mismo, cuándo debe ejecutarse la función *InicializarMapa*.

Mientras un navegador carga una página HTML y la construye, se obtienen, a la vez, imágenes externas y otros *scripts* que puedan formar parte de la página. Durante este proceso de carga, el entorno puede no ser lo bastante estable para Google Maps, y ello podría conducir a un comportamiento errático de la aplicación. Google recomienda inicializar el mapa sólo cuando la página haya sido cargada completamente.

Para ello, utilizaremos el atributo "onload" del elemento HTML *body*. En él podemos introducir código JavaScript que se ejecutará cuando la página haya sido cargada completamente. En nuestro caso, pondremos una llamada a la función *InicializarMapa*:

```
<body onload="InicializarMapa()" onunload="GUnload()">
```

De la misma forma, cuando se descarga la página, cosa que sucede cuando se cambia a otra, llamaremos a la función *GUnload* de la API de Google Maps para liberar correctamente los recursos que haya consumido el mapa.

El resultado será el siguiente:



2.2.2. Tipos de mapas

De forma predeterminada, la API de Google Maps muestra el mapa de calles y carreteras. Aunque el tipo de mapa puede cambiarse mediante los controles situados en su parte superior derecha, a veces es deseable mostrar uno u otro tipo desde buen comienzo.

Podemos cambiar el tipo de mapa mediante la función *setMapType* de la clase "GMap2". La función espera un solo parámetro, el tipo de mapa, que ha de ser uno de los siguientes:

- **G_NORMAL_MAP**, para el mapa de calles y carreteras,
- **G_SATELLITE_MAP**, para el mapa fotográfico,

- **G_HYBRID_MAP**, para el mapa híbrido (mapa fotográfico con las calles y las carreteras superpuestas), y
- **G_PHYSICAL_MAP**, para el mapa de relieve.

Siguiendo con el ejemplo anterior, si agregáramos la línea siguiente al final de la función *InicializarMapa*:

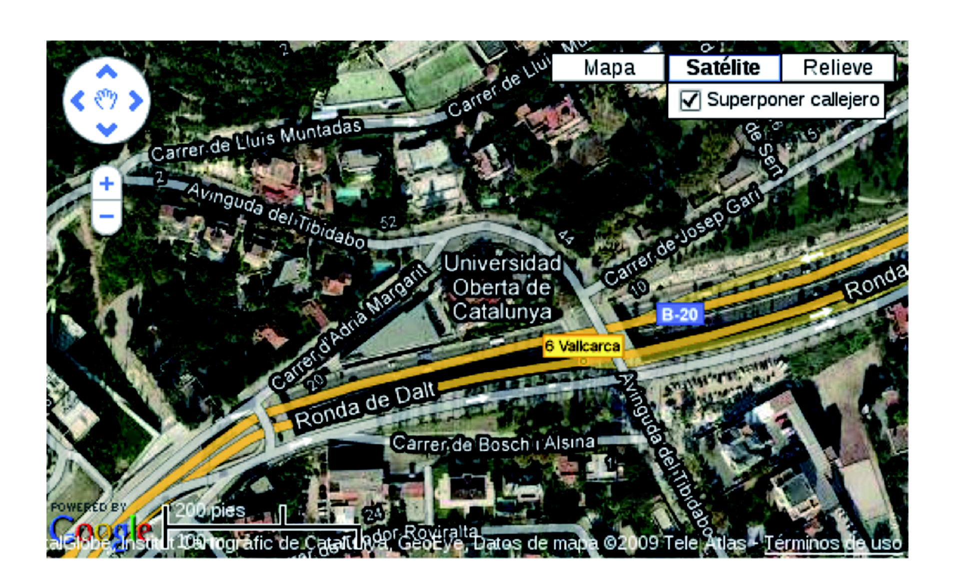
```
mapa.setMapType(G_HYBRID_MAP);
```

obtendríamos el mapa híbrido. El código completo de la función quedaría así:

```
<script type="text/javascript">
function InicializarMapa() {
  var mapa;
  var centro;

  if (GBrowserIsCompatible()) {
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    centro = new GLatLng(41.414829, 2.133003);
    mapa.setCenter(centro, 17);
    mapa.setUIToDefault();
    mapa.setMapType(G_HYBRID_MAP);
  }
}
</script>
```

y el resultado que obtendríamos sería el siguiente:



2.2.3. Transformación de coordenadas

Una funcionalidad importante que nos ofrecen las herramientas SIG es la de transformar las coordenadas de la pantalla en coordenadas geográficas y viceversa. Así, podemos determinar las coordenadas geográficas de un punto del mapa o situar en el mapa un punto geográfico determinado. Veamos cómo se manejan ambos casos con Google Maps.

Supongamos que queremos marcar una coordenada geográfica sobre el mapa. Para este propósito, Google Maps nos proporciona marcadores (“GMarker”) que se pueden emplazar en una posición determinada. El constructor de la clase “GMarker” espera un objeto “GLatLng” que indique la coordenada geográfica que deberá señalar.

El ejemplo siguiente destaca la situación del rectorado de la UOC en medio de Barcelona mediante el uso de un marcador:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
content="text/html; charset=utf-8"/>
<title>
Rectorado de la Universitat Oberta de Catalunya
</title>
<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>
<script type="text/javascript">
function InicializarMapa() {
  var mapa;
  var posicion_rectorado;
  var marca_rectorado;

  if (GBrowserIsCompatible()) {
    posicion_rectorado = new GLatLng(41.414829, 2.133003);

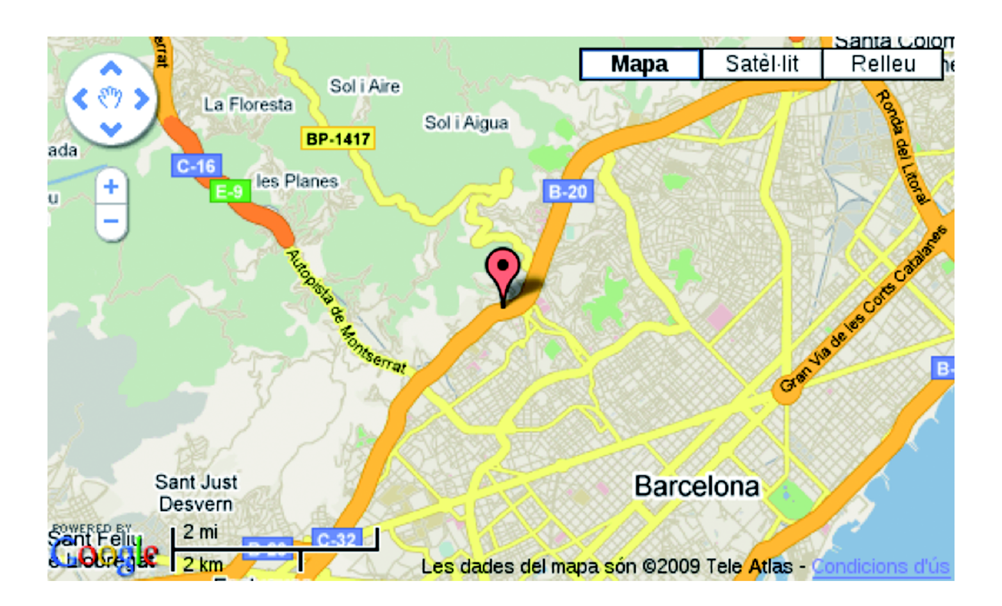
    // crea el mapa y lo inicializa
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(posicion_rectorado, 12);
    mapa.setUIToDefault();

    // crea un marcador sobre el rectorado de la UOC
    marca_rectorado = new GMarker(posicion_rectorado);
```

```
    mapa.addOverlay(marca_rectorado);
  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
</body>
</html>
```

Como se puede observar, en primer lugar se crea un objeto “GLatLng” con la posición del rectorado de la UOC: (41,414829N, 2,133003E). Este objeto es utilizado a posteriori para establecer el centro del mapa (función *setCenter*) y para crear el marcador. Finalmente, la función *addOverlay* (agregar capa) agrega el marcador al mapa.

El resultado será el siguiente:



Supongamos ahora que deseamos hacer el proceso inverso: determinar la posición geográfica de un punto del mapa. Será necesario recurrir a esta transformación cuando, por ejemplo, deseemos conocer las coordenadas geográficas de una posición del mapa sobre la que hayamos hecho clic.

En el ejemplo siguiente se muestran las coordenadas geográficas (expresadas en grados norte y este) del punto del mapa sobre el que se hace clic:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
<meta http-equiv="content-type"
content="text/html; charset=utf-8"/>
<title>Buscar coordenadas</title>
<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>
<script type="text/javascript">
function CuandoClic(capa, latlng, capaLatlng) {
  this.openInfoWindowHtml(latlng,
    latlng.lat() + "N<br>" + latlng.lng() + "E");
}
function InicializarMapa() {
  var centro_mapa;
  var mapa;

  if (GBrowserIsCompatible()) {
    centro_mapa = new GLatLng(41.414829, 2.133003);

    // crea el mapa y lo inicializa
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(centro_mapa, 15);
    mapa.setUIToDefault();

    // asocia los clics sobre el mapa con la función
    // "CuandoClic"
    GEvent.addListener(mapa, "click", CuandoClic);
  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
</body>
</html>
```

Para lograrlo, hemos asociado la función *CuandoClic* a los clics del ratón mediante la función *addListener* de la clase “GEvent”:

```
GEvent.addListener(mapa, "click", CuandoClic);
```

El primer parámetro es el mapa sobre el que se escucharán los clics; el segundo, el evento que se ha de capturar (en este caso el “click”, por el clic del ratón), y el último, la función a la que se llamará cada vez que se desate el evento.

Como hemos visto, cada vez que hagamos clic sobre el mapa se llamará a la función *CuandoClic*, que recibirá, por medio del parámetro “latlng” (de la clase “GLatLng”), las coordenadas geográficas sobre las que se ha apuntado.

La tarea de mostrar las coordenadas sobre el mapa la realiza la función *openInfoWindowHtml* del objeto “mapa”:

```
this.openInfoWindowHtml(latlng,
  latlng.lat() + "N<br>" + latlng.lng() + "E");
```

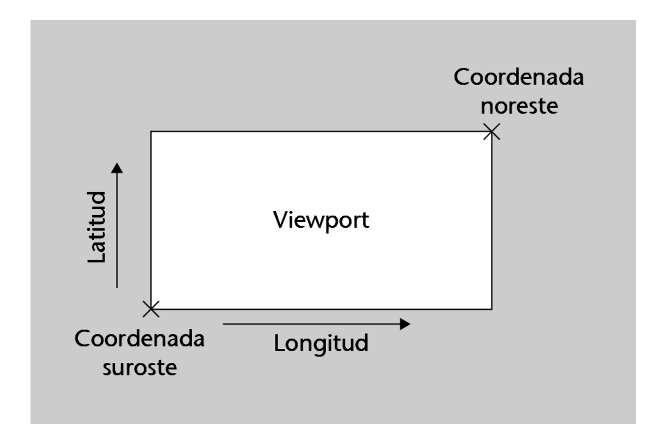
El primer parámetro son las coordenadas geográficas sobre las que apuntará el mensaje, y el segundo, el mensaje que se mostrará. El *this* se utiliza para hacer referencia al mapa actual.

Después de cada clic, obtendremos un resultado parecido al siguiente:



2.2.4. El *viewport*

Como ya hemos visto en el módulo anterior, el recuadro que delimita la parte visible del mapa recibe el nombre de *viewport*. También hemos visto que el *viewport* se caracteriza por las coordenadas geográficas de sus esquinas inferior izquierda (suroeste) y superior derecha (noreste), creciendo latitud y longitud desde la coordenada suroeste hasta la coordenada noreste.



Google Maps nos permite obtener los límites del *viewport* mediante la función *getBounds* de la clase "GMap2". Esta función devuelve un objeto de la clase "GLatLngBounds" que contiene las coordenadas geográficas de las esquinas suroeste y noreste del *viewport*. A su vez, dichas coordenadas las leeremos mediante las funciones *getLat* y *getLng*.

El ejemplo siguiente hace uso de la función *getBounds* para situar cinco marcas escogidas al azar dentro del *viewport*:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
content="text/html; charset=utf-8"/>
<title>Marcas aleatorias</title>
<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>
<script type="text/javascript">
function InicializarMapa() {
  var centro_mapa;
  var mapa;
  var limites;
  var noreste;
  var suroeste;
  var incrementoLatitud;
  var incrementoLongitud;
  var i;
  var marca;

  if (GBrowserIsCompatible()) {
    centro_mapa = new GLatLng(41.414829, 2.133003);
```

```
    // crea el mapa y lo inicializa
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(centro_mapa, 15);
    mapa.setUIToDefault();

    // obtiene los bordes noreste y sureste del 'viewport'
    limites = mapa.getBounds();

    noreste = limites.getNorthEast();
    suroeste = limites.getSouthWest();

    // obtiene la diferencia de latitud y longitud entre
    // ambos puntos
    incrementoLatitud = noreste.lat() - suroeste.lat();
    incrementoLongitud = noreste.lng() - suroeste.lng();

    // crea los cinco puntos, multiplicando las diferencias
    // por un número entre cero y uno (Math.random())
    i = 0;
    while (i < 5) {
      punto = new GLatLng(suroeste.lat() +
        incrementoLatitud * Math.random(),
        suroeste.lng() +
        incrementoLongitud * Math.random());
      marca = new GMarker(punto);
      mapa.addOverlay(marca);
      i++;
    }

  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
</body>
</html>
```

Como se puede observar, en un primer paso se obtienen los límites sureste y noreste del *viewport* mediante la función “getBounds”. Dichas coordenadas se almacenan en las variables “noreste” y “suroeste” respectivamente:

```
limites = mapa.getBounds();

noreste = limites.getNorthEast();
suroeste = limites.getSouthWest();
```

Hecho esto, en un segundo paso, se obtienen el ancho y el alto del *viewport*, restando por separado las latitudes y las longitudes de ambos puntos. Estas medidas, que representan el incremento máximo que pueden experimentar latitud y longitud, se almacenan en las variables “incrementoLatitud” e “incrementoLongitud” respectivamente:

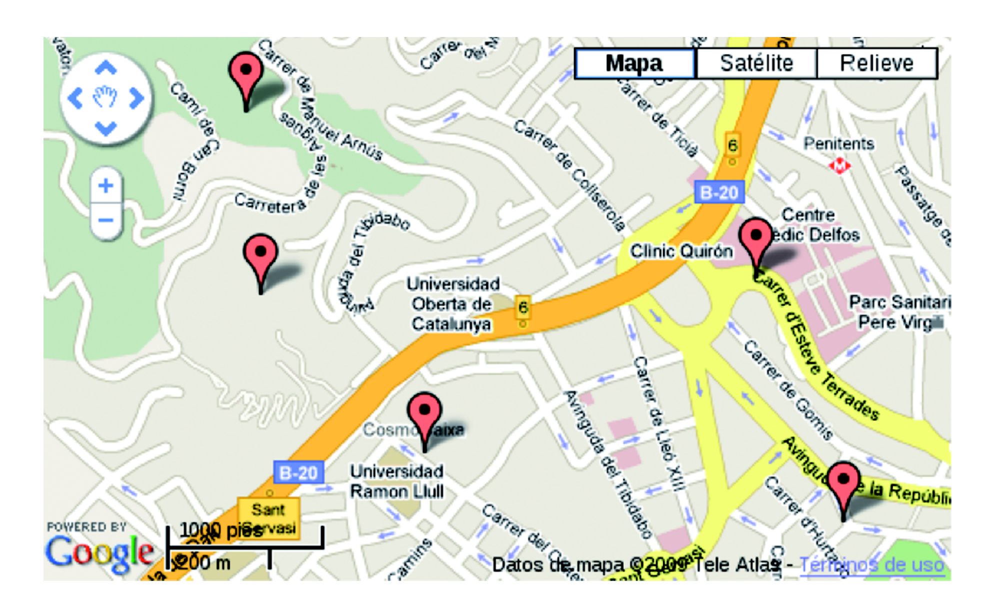
```
incrementoLongitud = noreste.lng() - suroeste.lng();
incrementoLatitud = noreste.lat() - suroeste.lat();
```

Ahora ya sólo queda sumar a la coordenada suroeste del *viewport* una latitud entre 0 e “incrementoLatitud” y una longitud entre 0 e “incrementoLongitud” para obtener una nueva coordenada dentro del *viewport*. Estos valores aleatorios se pueden obtener multiplicando “incrementoLatitud” e “incrementoLongitud” por un valor aleatorio entre 0 y 1, que es lo que proporciona la función *Math.random*:

```
punto = new GLatLng(suroeste.lat() +
  incrementoLatitud * Math.random(),
  suroeste.lng()
+ incrementoLongitud * Math.random());
```

Finalmente, se agregan las marcas al mapa mediante la función *addOverlay*.

El resultado será parecido al siguiente:



2.2.5. Líneas y polígonos

Google Maps nos permite representar objetos sobre un mapa, de manera que su geometría esté ligada a unas coordenadas geográficas determinadas. Buen ejemplo de ello son las marcas que hemos visto hasta ahora: se acomodan a los desplazamientos del mapa y a los cambios de zoom para apuntar siempre sobre el mismo lugar.

Sin embargo, las opciones que nos ofrece Google Maps no se acaban en los marcadores: también podemos representar líneas y polígonos.

Supongamos que queremos representar la ruta que hay que seguir para ir de Plaça de Catalunya de Barcelona al rectorado de la UOC. El código siguiente dibuja la ruta mediante un objeto "GPolyline"*:

*** Este ejemplo sólo pretende mostrar cómo trazar líneas con un objeto "GPolyline". Google Maps tiene mecanismos propios para trazar rutas entre dos puntos automáticamente. Si nuestro objetivo es obtener una ruta por carretera para ir de una dirección a otra, deberíamos usar la clase "GDirections".**

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
content="text/html; charset=utf-8"/>
<title>Cómo llegar a la UOC</title>
<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>
<script type="text/javascript">

function InicializarMapa() {
  var centro_mapa;
  var mapa;
  var ruta;

  if (GBrowserIsCompatible()) {
    centro_mapa = new GLatLng(41.38693554508235,
      2.169950008392334);

    // crea el mapa y lo inicializa
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(centro_mapa, 15);
    mapa.setUIToDefault();

    // dibuja la ruta
    ruta = new GPolyline(
      [ centro_mapa,
      new GLatLng(41.38799806199135, 2.1700572967529297),
```

```
      new GLatLng(41.39651366867027, 2.1594786643981934),
      new GLatLng(41.39981336757232, 2.1555089950561523),
      new GLatLng(41.40201843881663, 2.1529340744018555),
      new GLatLng(41.403965921268934, 2.1518611907958984),
      new GLatLng(41.40649273299253, 2.1495437622070312),
      new GLatLng(41.408279145680964, 2.1474623680114746),
      new GLatLng(41.41122420530506, 2.14508056640625),
      new GLatLng(41.41177136014642, 2.140810489654541),
      new GLatLng(41.415118560018506, 2.136197090148926),
      new GLatLng(41.41589096626463, 2.1361541748046875),
      new GLatLng(41.41479672137174, 2.1332359313964844) ],
      "#ff0000", 10, 0.5);
    mapa.addOverlay(ruta);
  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
</body>
</html>
```

Como se puede observar, el constructor de la clase “GPolyline” espera un vector con los puntos geográficos que constituyen la línea, su color, expresado en notación hexadecimal* (“#ff0000” por rojo), y su anchura, medida en píxeles (10). El cuarto parámetro, que es opcional, indica el grado de opacidad de la línea (0: transparente; 1: opaca). Observad que en JavaScript basta con encerrar una lista de objetos entre corchetes para crear un vector.

*** El código de color se puede obtener de cualquier herramienta de dibujo o edición fotográfica. Si no, hay paletas de colores disponibles en la Web (por ejemplo, en Wikipedia).**
http://es.wikipedia.org/wiki/Colores_HTML#Tabla_de_colores.

El resultado será el siguiente:



Basta con seguir la línea roja para llegar a nuestro destino.

Es importante destacar que, de forma predeterminada, las líneas dibujadas sobre un mapa mediante la clase "GPolyline" son líneas rectas respecto a la proyección, no respecto a la realidad. Esto es así porque la Tierra es curva, y para representarla sobre un mapa debemos deformarla ligeramente. Si la línea que queremos dibujar es suficientemente larga (de varios kilómetros), lo que sobre el mapa es recto, sobre el terreno será más o menos curvo y viceversa.

Para dibujar lineas geodésicas (aquellas que representan la distancia más corta entre dos puntos de la superficie terrestre), debemos pasarle al constructor de la clase "GPolyline" la opción *geodesic: true*. El ejemplo siguiente dibuja dos líneas entre Moscú y Pekín: la más corta sobre el mapa (en rojo) y la más corta sobre el terreno (en azul):

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
content="text/html; charset=utf-8"/>
<title>Moscú - Pekín</title>
<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>
<script type="text/javascript">
function InicializarMapa() {
  var centro_mapa;
  var mapa;
  var moscu;
  var pekin;
  var recta_mapa;
  var recta_terreno;

  if (GBrowserIsCompatible()) {
    centro_mapa = new GLatLng(45.9511496866914,
      79.1015625);

    // crea el mapa y lo inicializa
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(centro_mapa, 2);
    mapa.setUIToDefault();

    moscu = new GLatLng(55.7516682526483,
      37.61860370635986);
    pekin = new GLatLng(39.91582588231232,
      116.39079093933105);
```

```
    // dibuja la línea recta respecto a la proyección
    recta_mapa = new GPolyline([ moscu, pekin ],
      "#ff0000", 10, 0.5);
    mapa.addOverlay(recta_mapa);
    // dibuja la línea recta respecto al terreno
    recta_terreno = new GPolyline([ moscu, pekin ],
      "#0000ff", 10, 0.5, {geodesic: true});
    mapa.addOverlay(recta_terreno);
  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
</body>
</html>
```

El resultado será el siguiente:



De entre todas las funciones que nos ofrece la clase "GPolyline", quizás la más interesante sea la *getLength*. Esta función nos permite obtener la distancia real en metros de una línea, independientemente de si se ha representado como geodésica o no.

Aplicada a los objetos "recta_mapa" y "recta_terreno" del ejemplo anterior, en ambos casos la función "getLength" devolverá 5798434,48 m:

```
recta_mapa.getLength();    // devuelve 5798434,48
recta_terreno.getLength(); // devuelve 5798434,48
```

Dibujar polígonos

Aunque con un objeto “GPolyline” sería posible dibujar un polígono (en el caso de que el último punto fuera igual al primero), Google Maps nos ofrece una clase especializada para este propósito: “GPolygon”.

Al igual que con las líneas, podemos definir el color, el ancho y la opacidad del borde, además de poder definir el color y la opacidad del relleno.

Las líneas siguientes calculan el área de la Plaça de Catalunya de Barcelona:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
content="text/html; charset=utf-8"/>
<title>Área de Plaça de Catalunya</title>
<script src="http://maps.google.com/maps?file=api ↵
&amp;v=2&amp;key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
type="text/javascript"></script>
<script type="text/javascript">
function InicializarMapa() {
  var centro_mapa;
  var mapa;
  var poligono;

  if (GBrowserIsCompatible()) {
    centro_mapa = new GLatLng(41.3870240888213,
      2.1699607372283936);

    // crea el mapa y lo inicializa
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(centro_mapa, 16);
    mapa.setUIToDefault();

    // dibuja el área
    poligono = new GPolygon(
      [ new GLatLng(41.38787732230797, 2.169971466064453),
      new GLatLng(41.38741850946622, 2.168748378753662),
      new GLatLng(41.38585691167221, 2.1701431274414062),
      new GLatLng(41.38691944620778, 2.1712052822113037),
      new GLatLng(41.38787732230797, 2.169971466064453) ],
        "#ff0000", 10, 0.5, "#ff0000", 0.2);
    mapa.addOverlay(poligono);
```

```
    // muestra el área del polígono
    mapa.openInfoWindowHtml(centro_mapa,
      Math.round(poligono.getArea()) + " m<sup>2</sup>");
  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
style="width: 500px; height: 300px"></div>
</body>
</html>
```

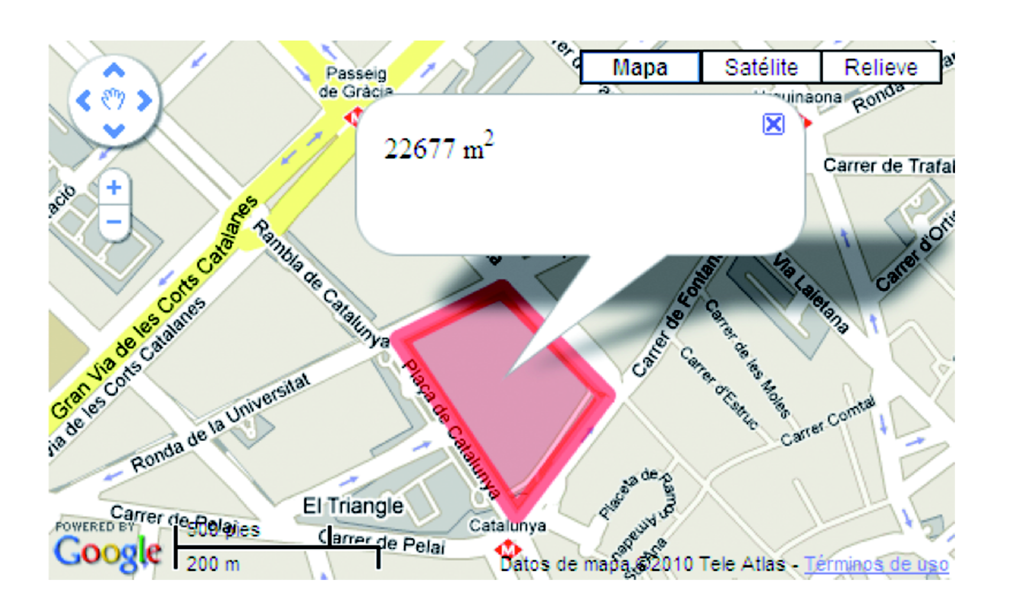
Como se puede observar, el constructor de la clase "GPolygon" es muy parecido al de la clase "GPolyline". Los tres primeros parámetros son idénticos: vector de puntos, color y opacidad del borde. Los dos parámetros que siguen son el color y la opacidad del relleno. Observad que el primer y el último de los puntos del vector son iguales. Google recomienda que sea así para evitar "comportamientos inesperados"*.

* Cita textual de la *Maps API Developer's Guide:* http://code.google.com/intl/ca/apis/maps/documentation/javascript/v2/overlays.html#Polygons_Overview

La última línea utiliza la función *getArea* para obtener el área, en metros cuadrados, del polígono. Observad que se utiliza la función *round* de la clase "Math" para redondear el resultado. El área obtenida se muestra por pantalla mediante la función *openInfoWindowHtml*:

```
mapa.openInfoWindowHtml(centro_mapa,
  Math.round(poligono.getArea()) + " m<sup>2</sup>");
```

El resultado será el siguiente:



Resumen

Después de aprender a programar aplicaciones SIG de escritorio, en este módulo se ha hecho lo propio con aplicaciones web. Si en el caso de las aplicaciones de escritorio nos hemos basado en un sistema ya implementado (gvSIG), con las aplicaciones web hemos hecho algo parecido: hemos utilizado la API pública de Google Maps.

Antes de abordar la estructura y las funciones de la API, hemos aprendido a crear una página web simple y a incorporarle un mapa. Después, hemos utilizado las funciones de la API para personalizar la presentación del mapa (coordenadas iniciales, zoom y tipo de mapa) y representar puntos, y hemos aprendido a transformar las coordenadas del mapa a coordenadas geográficas.

Finalmente, se han estudiado las funciones que permiten dibujar líneas y polígonos, caracterizando rutas y áreas que, más tarde, mediante funciones dedicadas, hemos podido medir.

Aunque todos los ejemplos están basados en la API de Google Maps, las técnicas explicadas en este módulo se pueden extrapolar más o menos directamente a otras API, libres o comerciales.

Bibliografia

Pere Barnola Augé, "*Introducción a la creación de páginas web*", http://mosaic.uoc.edu/2009/11/20/introduccion-a-la-creacion-de-paginas-web/, 09/2008.

W3C, "*HTML 4.01 Specification*", http://www.w3.org/TR/1999/REC-html401-19991224, 24/12/1999.

Varios autores, "*Parque nacional Hoge Veluwe*", http://es.wikipedia.org/wiki/Parque_nacional_Hoge_Veluwe.

Google, "*Google Maps API Concepts*", http://code.google.com/intl/es/apis/maps/documentation/javascript/v2/basics.html.

Google, "*Google Maps JavaScript API V2 Reference*", http://code.google.com/intl/es/apis/maps/documentation/javascript/v2/reference.html.

Uso de servicios OGC

Albert Gavarró Rodríguez

PID_00174758



Índice

Introducción

En los módulos anteriores, hemos aprendido a programar herramientas SIG utilizando siempre la cartografía que acompañaba a la aplicación: en el caso de GoogleMaps, los cuatro tipos de mapa que nos ofrece su API, y en el caso de gvSIG, los mapas que habíamos obtenido previamente de algun servidor de mapas.

Sin embargo, la proliferación de los servicios de cartografía en línea ha hecho posible una nueva forma de trabajar en la que el cliente no dispone ya de una copia propia de los datos, sino que accede continuamente a servicios en línea que mantienen una copia actualizada de esos datos.

En este módulo, aprenderemos a utilizar la cartografía disponible en línea, es decir, a interaccionar con los servicios OGC desde nuestro código para obtener todo tipo de mapas e incorporarlos a nuestra aplicación SIG.

Concretamente, aprenderemos a utilizar un servicio OGC/WMS desde una aplicación web basada en la API de Google Maps, de la misma manera que lo hacen aplicaciones como Wikiloc*. También aprenderemos a combinar las fuentes de datos de Google con los mapas obtenidos mediante el servicio WMS, e incluso prescindiremos de los mapas de Google para combinar dos o más mapas WMS externos.

*** http://www.wikiloc.com. Wikiloc es una aplicación web que permite compartir rutas georreferenciadas.**

Aunque este módulo trata, exclusivamente, sobre el servicio OGC/WMS, las técnicas aquí explicadas son fácilmente trasladables a los demás servicios OGC que proporcionan datos en forma de imágenes.

Objetivos

Al final de este módulo, deberíamos ser capaces de:

1. Crear tipos de mapa personalizados en Google Maps.

2. Agregar nuevos tipos de mapa a Google Maps.

3. Llamar a un servicio OGC/WMS desde una aplicación web.

4. Crear mapas personalizados en GoogleMaps que incorporen datos procedentes de servicios WMS.

5. Combinar los mapas de Google Maps con mapas WMS externos.

6. Combinar dos o más fuentes WMS prescindiendo de los mapas de Google Maps.

1. Google Maps y WMS

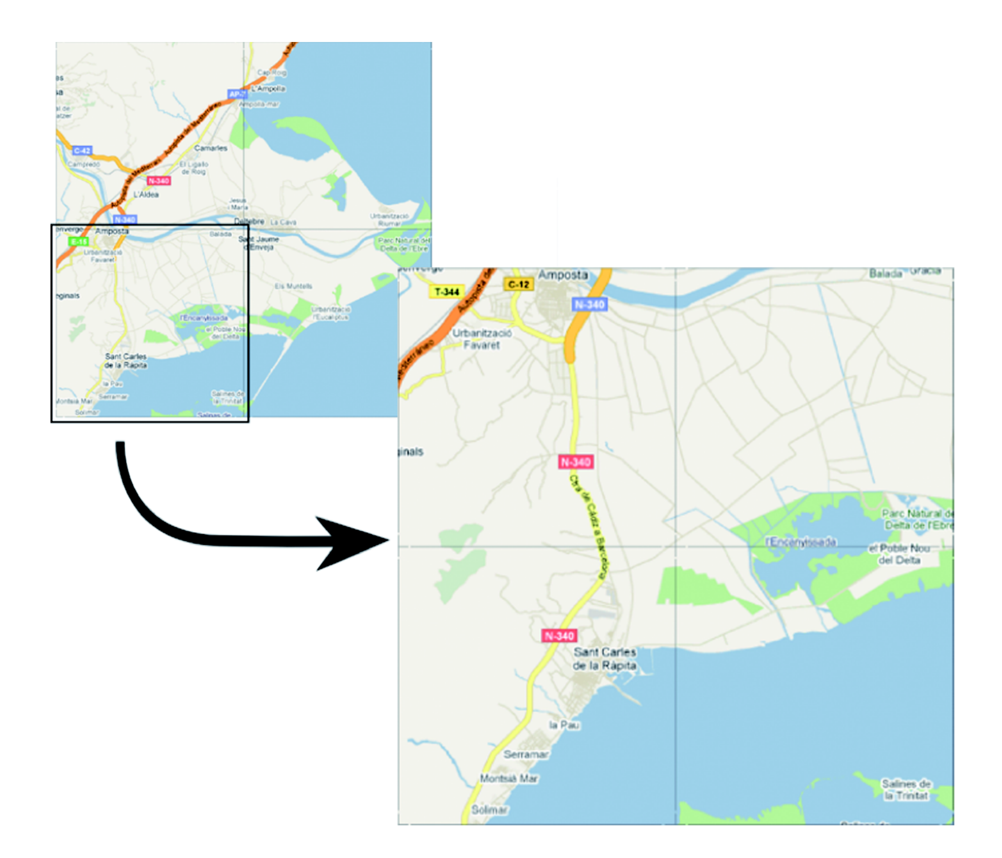
1.1. Anatomía de un mapa: los *tiles*

Todos los mapas de Google Maps estan divididos en porciones de 256 × 256 píxeles llamadas *tiles*. Cuando se solicita un mapa, Google Maps descarga del servidor las distintas porciones por separado y las junta, creando la ilusión de una única imagen:



El objetivo de reducir el mapa en porciones no es otro que minimizar el tamaño de los datos que el servidor tiene que enviar a las distintas aplicaciones. Debemos pensar que si bien en niveles de *zoom* pequeños sí que sería factible tener todo el mapa en una única imagen (cuyo tamaño sería de alrededor de unos pocos kilobytes), no sucede lo mismo con los niveles de *zoom* grandes, en los que sería prácticamente imposible enviar el mapa entero (cuyo tamaño sería de gigabytes).

En el nivel de *zoom* más pequeño (el nivel 0), el planeta entero está representado en una única porción. Cada nivel subsiguiente de *zoom* divide cada porción en cuatro subporciones, de manera que el número total de porciones responde a la fórmula 4^N, donde N es el nivel de *zoom*. Así pues, en el nivel 1 habrá cuatro porciones (que formarán una rejilla de 2 × 2), en el nivel 2, dieciséis (que formarán una rejilla de 4 × 4), y así sucesivamente.



Dependiendo del tamaño del *viewport*, la coordenada central del mapa y el nivel de *zoom*, se mostrarán unas porciones u otras.

1.2. Mapas personalizados

La API de Google Maps nos permite definir mapas personalizados, cuyo contenido (el contenido de cada una de las porciones) podemos elegir arbitrariamente. Una vez creados, estos mapas se pueden agregar a la vista mediante dos mecanismos diferenciados:

- como una capa del mapa actual, o
- como un mapa independiente.

Aunque existan dos formas de agregar mapas personalizados a una vista, la forma de implementarlos es siempre la misma: crear un objeto "GTileLayer" y sobrecargar las funciones siguientes:

- ***getTileUrl***. Devuelve la dirección (URL) que contiene la imagen de la porción (*tile*) del mapa, dadas las coordenadas de la porción y el nivel de *zoom*.

- ***isPng***. Devuelve un valor booleano indicando si la imagen de la porción está en formato PNG*, en cuyo caso puede contener transparencias.

- ***getOpacity***. Devuelve un valor entre 0,0 y 1,0 indicando el nivel de opacidad del mapa. 0,0 es completamente transparente, y 1,0, completamente opaco.

* PNG son las siglas de *Portable Network Graphics* (gráficos de red portables). El formato PNG es un formato gráfico basado en un algoritmo de compresión sin pérdida no sujeto a patentes. http://tools.ietf.org/html/rfc2083

Ejemplo 1: añadir una marca de agua a un mapa

Veamos un ejemplo sencillo de implementación de un mapa personalizado. Supongamos que deseamos mostrar los mapas de Google con el logotipo de la UOC sobreimpreso a modo de marca de agua*.

*** Una marca de agua es una imagen translúcida que se muestra sobre otra imagen o texto y que sirve para identificar el origen del documento.**

La idea es que la marca de agua se vaya repitiendo a lo largo y ancho del mapa, a intervalos regulares. Para lograr este objetivo, emplazaremos una misma imagen —el logotipo de la UOC— en todas y cada una de las porciones que, recordemos, dividen el mapa en áreas de 256 × 256 píxeles. A todos los efectos, es como si creásemos un mosaico con el logotipo de la UOC. Para crear el efecto de marca de agua, estableceremos el nivel de opacidad del mapa en el 50%.

Suponiendo que tenemos el logotipo de la UOC en un fichero llamado “logo-uoc.png” (de 256 × 256 píxeles), el código que crea el mapa nos quedaría así:

```
var capaLogo;

capaLogo = new GTileLayer(null, 0, 19);

// el nombre del fichero que contiene la imagen
capaLogo.getTileUrl = function() { return "logo-uoc.png"; }

// opacidad del 50%, expresada en tanto por uno
capaLogo.getOpacity = function() { return 0.5; }

// el fichero que contiene la imagen está en formato PNG
capaLogo.isPng = function() { return true; }
```

Como podemos observar, el código sobrecarga las tres funciones de la clase “GTileLayer” antes descritas: *getTileUrl*, *getOpacity* e *isPng*. La función *getTileUrl* siempre devuelve la misma imagen (“logo-uoc.png”), que es la que queremos mostrar en todas las porciones del mapa; *getOpacity*, siempre la misma transparencia (del 50%), e *isPng*, siempre *true*, puesto que sabemos que la imagen que se devuelve está en formato PNG. Por otro lado, el constructor de la clase “GTileLayer” espera tres parámetros:

- una colección de textos de *copyright* que se mostrarán en la parte inferior derecha del mapa,
- el nivel de *zoom* mínimo del mapa (ampliación mínima del mapa) y
- el nivel de *zoom* máximo del mapa (ampliación máxima del mapa).

En el ejemplo, no se ha proporcionado ninguna colección de textos de *copyright* (“null”) y se han establecido los niveles de *zoom* mínimo y máximo permitidos por Google Maps.

En los apartados siguientes, aprenderemos a agregar este mapa a la vista.

1.2.1. *Tile layer overlays*

Una forma de agregar un mapa personalizado a la vista es mediante el mecanismo de las capas. Google llama a las capas que contienen mapas *tile layer overlays*.

Por medio de este mecanismo, situaremos el mapa en la capa más superficial de la vista, de manera que quede por encima de las demás. Dada su posición predominante, debemos jugar con su opacidad para que revele, en mayor o menor medida, las capas inferiores.

Los pasos que se habrán de seguir serán dos:

1) crear una capa a partir del objeto "GTileLayer" que contiene el mapa, y
2) agregar dicha capa a la vista.

Siguiendo con el ejemplo anterior, el código siguiente:

```
mapa.addOverlay(new GTileLayerOverlay(capaLogo));
```

crea una capa a partir del mapa “capaLogo” y lo agrega a la vista del mapa.

Como se puede observar, el primer paso se realiza instanciando un objeto "GTileLayerOverlay", cuyo constructor espera un objeto "GTileLayer", es decir, el mapa personalizado ("capaLogo"). El segundo paso se alcanza mediante una llamada a la función *addOverlay* de la clase "GMap2", cuyo único parámetro debe ser un objeto "GTileLayerOverlay", es decir, la capa que se desea agregar a la vista.

Debemos notar que estamos agregando el mapa a la vista, del mismo modo en que se agrega un marcador. En consecuencia, tanto el mapa como el marcador se mostrarán en el primer plano de la vista.

Juntándolo todo, el código nos quedaría así:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
  content="text/html;charset=utf-8"/>
<title>Ejemplo GTileLayer</title>
<script
  src="http://maps.google.com/maps?file=api&amp;v=2"
  type="text/javascript">
</script>
<script type="text/javascript">
function InicializarMapa() {
  var mapa;
  var capaLogo;

  if (GBrowserIsCompatible()) {
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(new GLatLng(41.5580, 1.5906), 7);
    mapa.setUIToDefault();

    capaLogo = new GTileLayer(null, 0, 19);

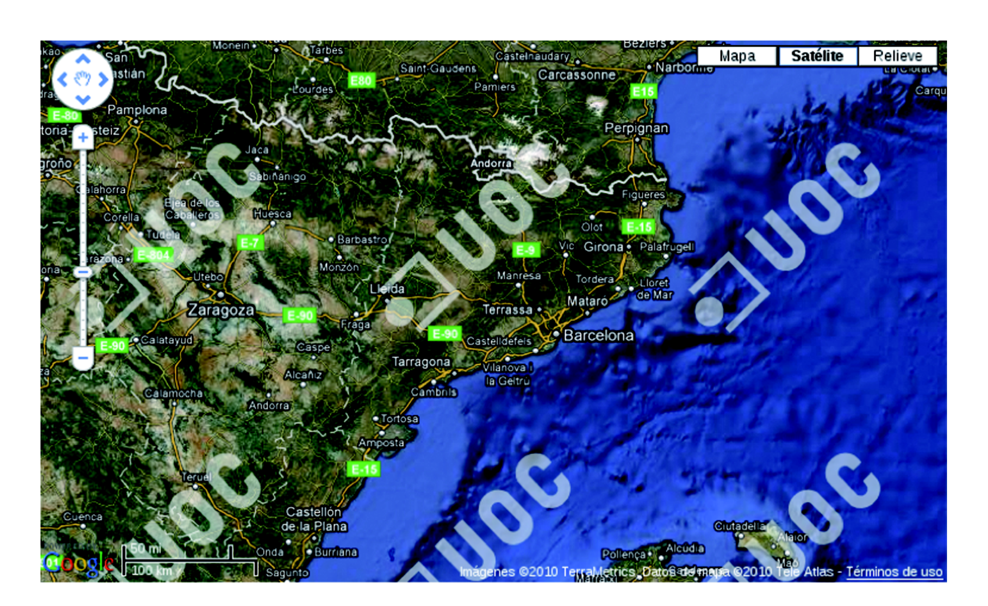
    // el nombre del fichero que contiene la imagen
    capaLogo.getTileUrl = function() {
```

```
      return "logo-uoc.png";
    }
    // opacidad del 50%, expresada en tanto por uno
    capaLogo.getOpacity = function() { return 0.5; }

    // el fichero que contiene la imagen está en formato PNG
    capaLogo.isPng = function() { return true; }

    mapa.addOverlay(new GTileLayerOverlay(capaLogo));
  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
  style="width: 750px; height: 450px"></div>
</body>
</html>
```

El resultado será el siguiente:



1.2.2. Tipos de mapa personalizados

Otra forma de agregar un mapa personalizado a una vista es creando un tipo de mapa personalizado. Mediante este mecanismo, mostraremos el nuevo mapa aislado de los demás y podremos acceder a él mediante un botón dedicado que se mostrará al lado de los botones de los tipos de mapa predeterminados.

Los pasos que se habrán de seguir serán dos:

1) crear un tipo de mapa personalizado, definiendo el contenido de sus capas, y
2) agregar el nuevo tipo de mapa a la vista.

Partiendo otra vez del mapa que hemos creado con el mosaico de logotipos de la UOC, el código siguiente:

```
var mapaLogo;

mapaLogo = new GMapType([capaLogo],
   G_NORMAL_MAP.getProjection(), "Marca de agua");
mapa.addMapType(mapaLogo);
```

crea un nuevo tipo de mapa (llamado “Marca de agua”) y lo agrega a la vista del mapa.

Como se puede observar, el primer paso se realiza instanciando un objeto de la clase “GMapType”, cuyo constructor espera tres parámetros: un vector con las capas que contiene el mapa, la proyección del mapa y el nombre del tipo de mapa. Debemos observar que el tipo de mapa que acabamos de crear sólo tendrá una capa —la del mosaico del logotipo de la UOC, “capaLogo”—, y tendrá la misma proyección que el mapa normal (“G_NORMAL_MAP”), que obtenemos mediante la función *getProjection*.

El segundo paso se realiza mediante la función *addMapType* de la clase “GMap2”, que espera un objeto “GMapType”.

Juntándolo todo, el código nos quedaría así:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
  content="text/html;charset=utf-8"/>
<title>Ejemplo GTileLayer</title>
<script
  src="http://maps.google.com/maps?file=api&amp;v=2"
  type="text/javascript">
</script>
<script type="text/javascript">
function InicializarMapa() {
  var mapa;
  var capaLogo;
  var mapaLogo;

  if (GBrowserIsCompatible()) {
    mapa = new GMap2(
      document.getElementById("elemento_mapa"));
    mapa.setCenter(new GLatLng(41.5580, 1.5906), 7);
```

```
    mapa.setUIToDefault();

    capaLogo = new GTileLayer(null, 0, 19);

    // el nombre del fichero que contiene la imagen
    capaLogo.getTileUrl = function() {
      return "logo-uoc.png";
    }

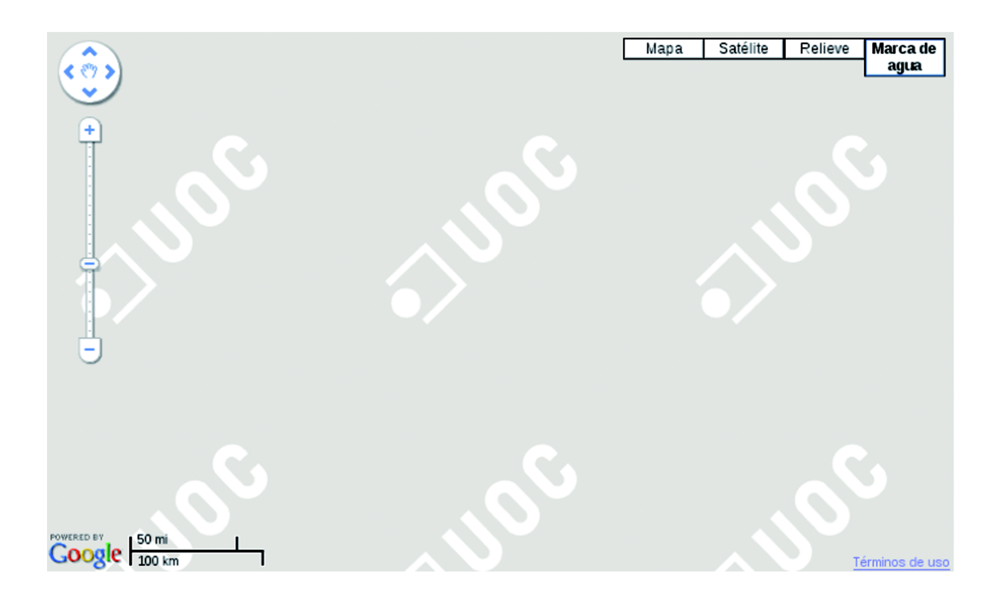
    // opacidad del 100%, expresada en tanto por uno
    capaLogo.getOpacity = function() { return 1.0; }

    // el fichero que contiene la imagen está en formato PNG
    capaLogo.isPng = function() { return true; }

    // crea un nuevo tipo de mapa
    mapaLogo = new GMapType([capaLogo],
      G_NORMAL_MAP.getProjection(),
      "Marca de agua");
    mapa.addMapType(mapaLogo);
  }
}
</script>
</head>
<body onload="InicializarMapa()" onunload="GUnload()">
<div id="elemento_mapa"
  style="width: 750px; height: 450px"></div>
</body>
</html>
```

Como en esta ocasión la capa que contiene el mosaico se muestra aislada, hemos cambiado su opacidad a 1,0, de manera que sea completamente opaca.

El resultado será el siguiente:



1.3. Mapas WMS

Hasta ahora hemos aprendido a crear mapas personalizados y a agregarlos, como capas o como mapas aislados, a la vista de mapa. Sin embargo, el contenido que mostrábamos era muy poco interesante: tan sólo una imagen que se repetía una y otra vez.

Ahora utilizaremos esta base para crear mapas personalizados que muestren datos obtenidos de un servicio OGC/WMS. La idea que reside en el fondo de esta técnica es relativamente simple: obtener la imagen de cada porción de un servicio OGC/WMS y proporcionarla a Google Maps.

1.3.1. El servicio OGC/WMS

El servicio OGC/WMS, en adelante WMS (por *Web Map Service,* servicio de mapas de la Web), proporciona mapas georeferenciados a partir de datos geográficos. Los mapas se generan a petición de un cliente, que especifica, mediante un protocolo estándar*, los datos que quiere obtener, la zona geográfica en la que se limita la petición, el formato del mapa obtenido, etc. Al final, el cliente obtiene una imagen georeferenciada que concentra y representa toda la información que esperaba obtener. Es importante destacar que el cliente no recibe nunca los datos en sí: tan sólo una imagen apta (hablando en términos de resolución) para ser mostrada en la pantalla de un ordenador.

*** Se puede consultar la especificación del protocolo en: http://www.opengeospatial.org/standards/wms**

El estándar define dos operaciones obligatorias que debe soportar el servicio (*GetCapabilities* y *GetMap*) y otra optativa (*GetFeatureInfo*):

- ***GetCapabilities***. Proporciona información acerca de los datos ofrecidos por el servidor y sus características, los sistemas de referencia soportados, el ámbito geográfico, etc.

- ***GetMap***. Proporciona una imagen georeferenciada que contiene los datos solicitados. El estándar nos permite definir el área del mapa, las capas mostradas, el formato de la imagen, etc.

- ***GetFeatureInfo***. Proporciona información acerca de características particulares mostradas en el mapa.

La información que se puede obtener por medio del servicio WMS gira en torno a las capas de información. Es decir, el proveedor proporciona una o más capas, que se pueden obtener aisladas o combinadas mediante la operación *GetMap*. Es importante remarcar que siempre se obtiene una única imagen como respuesta a una operación *GetMap*.

A continuación, se detallan los parámetros obligatorios que espera la operación *GetMap* y el significado de cada uno de ellos:

- **VERSION=1.1.1.** Especifica el número de versión del protocolo. En otras palabras, especifica la versión del servicio a la que se espera acceder. Dependiendo de la versión, el proveedor aceptará unos parámetros u otros. Estos materiales se basan exclusivamente en la versión 1.1.1 del estándar.

- **REQUEST=GetMap.** Especifica la operación a la que se quiere acceder. Para la operación *GetMap*, el valor de REQUEST siempre será *GetMap*.

- **LAYERS (capas).** Especifica las capas de información que se desean obtener, separadas por comas. Las capas se deben listar desde la más profunda hasta la más superficial.

- **STYLES (estilos).** Especifica el estilo con el que se dibujará cada capa. Igual que en el parámetro anterior, los nombres de los estilos deben ir separados por comas.

- **SRS (de *Source Reference System*, 'sistema de referencia de la fuente').** Especifica el sistema de referencia de la petición. Todas las coordenadas especificadas en los demás parámetros se suponen escritas en este sistema de referencia.

- **BBOX (de *Bounding BOX*, 'cuadro delimitador').** Especifica los límites del mapa que se desea obtener. Se trata de una lista separada por comas de las coordenadas suroeste y noreste del mapa. Las coordenadas deben especificarse de acuerdo al sistema de referencia indicado en el parámetro SRS.

- **WIDTH (ancho).** Especifica la anchura, medida en píxeles, que debe tener la imagen resultante.

- **HEIGHT (alto).** Especifica la altura, medida en píxeles, que debe tener la imagen resultante.

- **FORMAT (formato).** Especifica el formato (PNG, JPEF, GIF, etc.) que debe tener la imagen resultante.

En resumen, mediante la operación *GetMap* podemos obtener una porción de cualquier mapa que nos ofrezca el proveedor, en el formato y en las dimensiones que deseemos.

1.3.2. Petición de datos

Toda la potencia y la flexibilidad que nos ofrece el servicio WMS queda ensalzada por otra característica no menos importante del servicio: su accesibilidad desde la Web. Como su nombre indica (*Web Map Service*), cualquier cliente de la Web puede conectarse a un servicio WMS y obtener mapas.

En este sentido, el estándar ha dispuesto que uno de los métodos de acceso a las operaciones WMS sea uno de los mecanismos fundamentales de la Web: el método HTTP GET. En otras palabras, se puede acceder a las operaciones mediante una URL cualquiera, de la misma forma que solicitamos una página web.

Así pues, si introducimos la URL siguiente en nuestro navegador:

```
http://shagrat.icc.es/lizardtech/iserv/
ows?SERVICE=WMS&REQUEST=GetMap&VERSION=1.1.1&LAYERS=mt
c50m&STYLES=&SRS=EPSG:23031&BBOX=290368.84,4538236.42,
292203.28,4540070.86&WIDTH=500&HEIGHT=500&FORMAT=JPEG
```

obtendremos un mapa topográfico a escala 1:50000 de Benifallet y sus alrededores.

Si observamos detenidamente la URL, veremos los parámetros que hemos descrito en el apartado anterior separados por el carácter *et* ("&", *ampersand* en inglés): REQUEST, SERVICE, VERSION, SRS, BBOX, WIDTH, HEIGHT, LAYERS, STYLES y FORMAT. La tabla siguiente muestra los valores asignados a cada uno de ellos y su significado:

Parámetro	Valor	Significado
SERVICE	WMS	Especifica que se desea acceder al servicio WMS.
REQUEST	GetMap	Especifica que se desea realizar la operación *GetMap*.
VERSION	1.1.1	Especifica que se desea utilizar la versión 1.1.1 del protocolo WMS.
LAYERS	mtc50m	Especifica que se desea acceder a los datos de la capa "mtc50m". Las capas serán exclusivas de cada servidor, y se podrá obtener una lista de ellas mediante la operación *GetCapabilities*. En este caso, se trata de un mapa topográfico a escala 1:50000 ofrecido por el ICC.
STYLES	vacío	Especifica que no se desea ningún estilo en particular. De hecho, la capa seleccionada no tiene ningún estilo asociado. De nuevo, los estilos son propios de cada servidor y de cada capa. Se puede obtener una lista de estilos mediante la operación *GetCapabilities*.
SRS	EPSG:23031	Especifica que las coordenadas utilizadas en los demás parámetros estarán escritas según el sistema de referencia European Datum 1950 proyección UTM Huso 31 Norte (EPSG:23031).
BBOX	290368.84, 4538236.42, 292203.28, 4540070.86	Especifica los límites del área que se desea obtener mediante las coordenadas de la esquina superior derecha (290368.84, 4538236.42) y la esquina inferior izquierda (292203.28, 4540070.86) del mapa. Recordemos que estas coordenadas están escritas en EPSG:23031.
WIDTH	500	Especifica que se desea obtener una imagen cuya anchura sea de 500 píxeles.
HEIGHT	500	Especifica que se desea obtener una imagen cuya altura sea de 500 píxeles.
FORMAT	JPEG	Especifica que se desea obtener una imagen en formato JPEG.

1.3.3. *GTileLayer WMS*

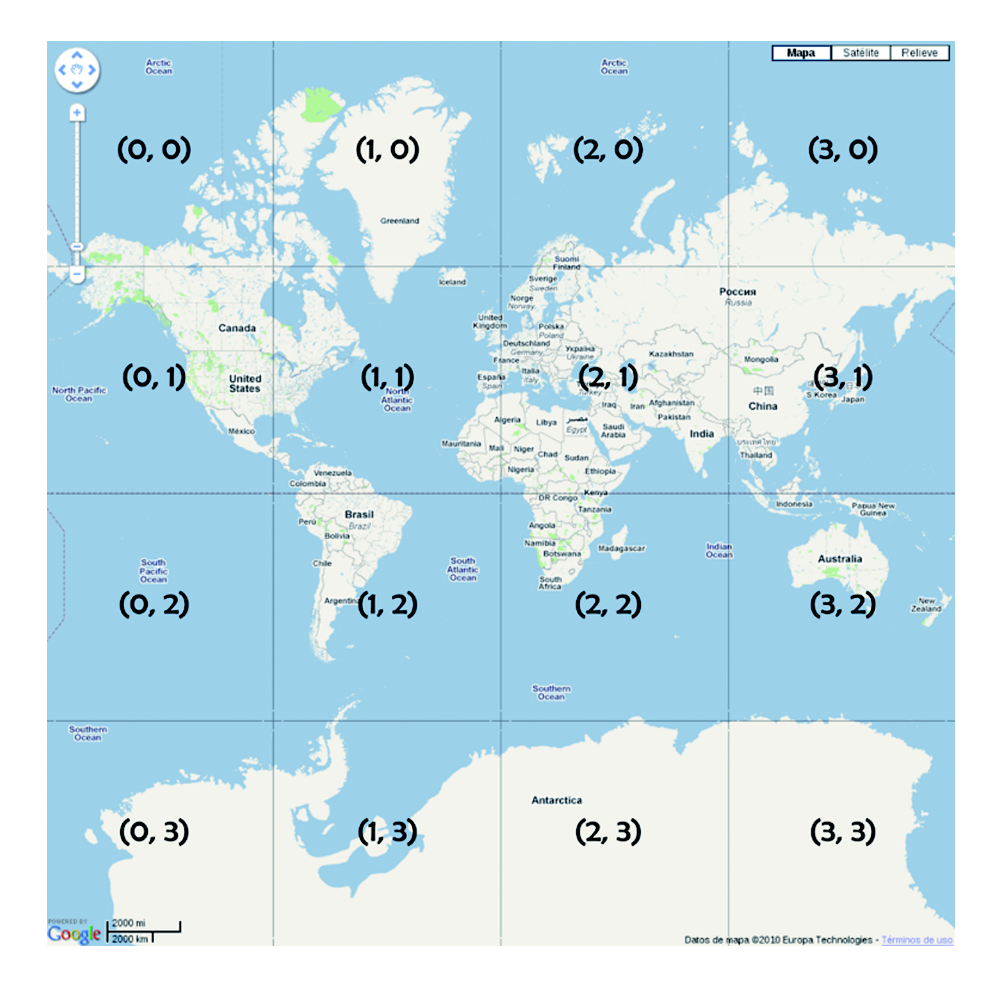
Ahora que ya sabemos cómo obtener una porción de un mapa mediante un servicio WMS, veamos cómo podemos trasladar estas porciones a un mapa personalizado de Google Maps.

Como ya hemos visto, un mapa de Google Maps se divide en porciones cuadradas de 256 × 256 píxeles. Además, el número de porciones varía en función del *zoom*, y obedece a la fórmula 4^N, donde N es el nivel de *zoom*. Así, en el nivel de *zoom* 0 (menor ampliación), tenemos una única porción; en el nivel 1, cuatro (formando una rejilla de 2 × 2); en el nivel 2, dieciséis (formando una rejilla de 4 × 4), y así sucesivamente.

Consecuentemente, podemos decir que en un mapa existen dos tipos de coordenadas:

- las coordenadas de un píxel dentro de una porción, y
- las coordenadas de una porción dentro del mapa.

Ambos tipos de coordenadas se expresan mediante dos componentes, x e y, donde x es el número de columna en la que se encuentra el píxel o la porción dentro de la rejilla, e y es el número de fila. En ambos casos, se empieza contando desde 0.



Cuando nos hemos introducido en la creación de mapas personalizados, hemos mencionado que la función *getTileUrl* de la clase "GtileLayer" recibe por parámetro las coordenadas de la porción y el nivel de *zoom*. Aunque entonces no ha sido necesario consultar ninguno de sus parámetros porque siempre se mostraba la misma imagen en todas las porciones, ahora ello sí que será necesario, puesto que el contenido de una porción u otra variará en función de la parte del mapa que se represente.

Para representar correctamente la parte del mapa correspondiente a una porción, debemos determinar el área que cubre dicha porción. Aquí ya empezamos a tener un punto de contacto con la función *GetMap* del servicio WMS, cuyo parámetro BBOX especificaba el área del mapa que se quería obtener mediante las coordenadas de las esquinas noreste y suroeste. Si somos capaces de obtener dichas coordenadas de una porción, podremos obtener la imagen del mapa por medio del servicio WMS.

Trasladadas al sistema de coordenadas de la API de Google Maps, las coordenadas noreste y sureste corresponden a las coordenadas de las esquinas superior izquierda e inferior derecha. Para obtenerlas, seguiremos los pasos siguientes:

1) calcular las coordenadas en píxeles de las esquinas de la porción, y
2) convertir las coordenadas en píxeles a coordenadas geográficas.

El primer paso lo resolveremos con unas pocas multiplicaciones y sumas. Dada una porción situada en la posición (x, y) de una rejilla isométrica en la que cada porción mide 256 × 256 píxeles, las coordenadas (v, w), expresadas en píxeles, de sus esquinas superior izquierda e inferior derecha responden a las fórmulas:

	Superior izquierda	**Inferior derecha**
v	x * 256	(x + 1) * 256
w	(y + 1) * 256	y * 256

El segundo paso lo realizaremos mediante la función *fromPixelToLatLng*, de la clase "GProjection". El objeto "GProjection" (hay un objeto "GProjection" asociado a cada mapa), que gestiona la transformación de coordenadas, lo obtendremos mediante la función *getProjection* de la clase "GMap2".

Recapitulando, dado un objeto llamado "tile" de tipo "GPoint" que representa las coordenadas (x, y) de una porción del mapa, y un nivel de *zoom* representado por la variable "zoom", las líneas siguientes calculan las coordenadas geográficas que debemos pasar al servicio WMS ("geoSupIzq" y "geoInfDer", por este orden) para obtener la imagen del mapa que debe mostrarse en la porción representada por el objeto "tile".

```
// obtiene las coordenadas x/y de las esquinas superior
// izquierda e inferior derecha
xySupIzq = new GPoint(tile.x * 256, (tile.y + 1) * 256);
xyInfDer = new GPoint((tile.x + 1) * 256, tile.y * 256);

// obtiene las coordenadas geográficas de las esquinas
// superior izquierda e inferior derecha
geoSupIzq = G_NORMAL_MAP.getProjection().
  fromPixelToLatLng(xySupIzq, zoom);
geoInfDer = G_NORMAL_MAP.getProjection().
  fromPixelToLatLng(xyInfDer, zoom);
```

Como se puede observar, la función *fromPixelToLatLng* transforma las coordenadas expresadas en píxeles en coordenadas geográficas. Dado que el número de porciones y, consecuentemente, las dimensiones del mapa varían con el nivel de *zoom*, la función *fromPixelToLatLng* necesita, además, que se le proporcione el nivel de *zoom* actual para poder realizar la transformación correctamente.

Ahora sólo nos queda construir la llamada a la función *GetMap* del servicio WMS con las coordenadas obtenidas. Las líneas siguientes:

```
// genera la URL del "tile"
urlTile  = wmsUrl;
urlTile += "&REQUEST=GetMap";
urlTile += "&SERVICE=WMS";
urlTile += "&VERSION=" + wmsVersion;
urlTile += "&LAYERS=" + wmsCapas;
urlTile += "&STYLES=" + wmsEstilos;
urlTile += "&FORMAT=" + wmsFormato;
urlTile += "&BGCOLOR=" + wmsColorFondo;
urlTile += "&TRANSPARENT=TRUE";
urlTile += "&SRS=" + this.wmsSrs;
urlTile += "&BBOX=" + geoSupIzq.x + "," + geoSupIzq.y +
  "," + geoInfDer.x + "," + geoInfDer.y;
urlTile += "&WIDTH=256";
urlTile += "&HEIGHT=256";
urlTile += "&reaspect=false";
```

almacenan en la variable “urlTile” la URL que nos proporcionará la imagen de la porción caracterizada por las coordenadas “geoSupIzq” y “geoInfDer”. Las variables “wmsUrl” (la dirección base del servidor), “wmsVersion”, “wmsCapas”, “wmsEstilos”, “wmsFormato” y “wmsColorFondo” podrán contener valores distintos según el servidor al que nos hayamos conectado y las necesidades que tengamos. El significado de cada uno de estos parámetros se ha explicado en el apartado “El servicio OGC/WMS”.

Juntándolo todo, la función *getTileUrl* de un “GTileLayer” ligado a un servicio WMS tendrá un aspecto parecido al siguiente:

```
// tile: GPoint
// zoom: Number
function obtenerUrlTile(tile, zoom) {
  // coordenada x/y de la esquina superior izquierda
  var xySupIzq;
  // coordenada geográfica de la esquina superior izquierda
  var geoSupIzq;
  // coordenada x/y de la esquina inferior derecha
  var xyInfDer;
  // coordenada geográfica de la esquina inferior derecha
  var geoInfDer;
  var urlTile;

  // obtiene las coordenadas x/y de las esquinas superior
  // izquierda e inferior derecha
  xySupIzq = new GPoint(tile.x * 256, (tile.y + 1) * 256);
  xyInfDer = new GPoint((tile.x + 1) * 256, tile.y * 256);

  // obtiene las coordenadas geográficas de las esquinas
  // superior izquierda e inferior derecha
  geoSupIzq = G_NORMAL_MAP.getProjection().
    fromPixelToLatLng(xySupIzq, zoom);
  geoInfDer = G_NORMAL_MAP.getProjection().
    fromPixelToLatLng(xyInfDer, zoom);

  // genera la URL del "tile"
  urlTile  = this.wmsUrl;
  urlTile += "&REQUEST=GetMap";
  urlTile += "&SERVICE=WMS";
  urlTile += "&VERSION=" + this.wmsVersion;
  urlTile += "&LAYERS=" + this.wmsCapas;
  urlTile += "&STYLES=" + this.wmsEstilos;
  urlTile += "&FORMAT=" + this.wmsFormato;
  urlTile += "&BGCOLOR=" + this.wmsColorFondo;
  urlTile += "&TRANSPARENT=TRUE";
  urlTile += "&SRS=" + this.wmsSrs;
  urlTile += "&BBOX=" + geoSupIzq.x + "," + geoSupIzq.y +
    "," + geoInfDer.x + "," + geoInfDer.y;
  urlTile += "&WIDTH=256";
  urlTile += "&HEIGHT=256";
  urlTile += "&reaspect=false";

  return urlTile;
}
```

1.3.4. Mapas personalizados

Partiendo del código desarrollado en el apartado anterior, a continuación mostraremos algunos ejemplos que combinan los mapas de Google con mapas WMS, o que incluso prescinden de los mapas de Google para mostrar un único mapa WMS o un par de ellos combinados. Para ilustrar estos ejemplos, hemos utilizado los servicios WMS que ofrecen el ICC* i el CREAF**.

*** Los datos proporcionados por el servicio WMS del ICC se detallan en la página:**
http://www.icc.es/web/content/es/prof/cartografia/fitxes_geoserveis.html
**** Los datos proporcionados por el servicio WMS del CREAF se detallan en la página:**
http://www.opengis.uab.es/

Ejemplo 2: un único mapa WMS

En este ejemplo, se crea un mapa personalizado que contiene un único mapa topográfico a escala 1:250000 procedente del servicio WMS del ICC:

```
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type"
  content="text/html;charset=utf-8"/>
<title>Ejemplo 2</title>
<script src="http://maps.google.com/maps?file=api&amp;v=2"
  type="text/javascript"></script>
<script type="text/javascript">
function initialize() {
  var mapa;
  var capaTopo;
  var mapaTopo;

  if (GBrowserIsCompatible()) {
    mapa = new GMap2(document.getElementById("elemento_mapa"));

    mapa.setCenter(new GLatLng(41.5580, 1.5906), 7);
    mapa.setUIToDefault();

    // crea la capa WMS
    capaTopo = crearCapaWms(7, 12, 1.0,
      "http://shagrat.icc.es/lizardtech/iserv/ows?",
      "mtc250m,");

    // crea un mapa personalizado a partir de la capa WMS
    mapaTopo = new GMapType(
      [capaTopo],
      G_NORMAL_MAP.getProjection(),
      "WMS");

    // agrega el mapa personalizado al visor
    mapa.addMapType(mapaTopo);
  }
}

function crearCapaWms(zoomMin, zoomMax, opacidad, url, capas,
  estilos, formato, colorFondo, version, srs) {
  var capaWms;

  capaWms = new GTileLayer(null, zoomMin, zoomMax);

  // rellena los parámetros no especificados
  if (! estilos) { estilos = ""; }
  if (! formato) { formato = "image/png"; }
  if (! version) { version = "1.1.1"; }
  if (! colorFondo) { colorFondo = "0xFFFFFF"; }
  if (! srs) { srs = "EPSG:4326"; }
```

```
  capaWms.wmsUrl = url;
  capaWms.wmsCapas = capas;
  capaWms.wmsEstilos = estilos;
  capaWms.wmsFormato = formato;
  capaWms.wmsVersion = version;
  capaWms.wmsColorFondo = colorFondo;
  capaWms.wmsSrs = srs;

  // la dirección web del fichero que contiene la imagen
  capaWms.getTileUrl = obtenerUrlTile;

  // opacidad expresada en tanto por uno
  capaWms.getOpacity = function() { return opacidad; }

  // el fichero que contiene la imagen está en formato PNG?
  capaWms.isPng = function() { return formato == "image/png"; }

  return capaWms;
}

// tile: GPoint
// zoom: Number
function obtenerUrlTile(tile, zoom) {
  var xySupIzq;  // coord. x/y de la esquina superior izquierda
  var geoSupIzq; // coord. geográfica de la esquina superior
                 // izquierda
  var xyInfDer;  // coord. x/y de la esquina inferior derecha
  var geoInfDer; // coord. geográfica de la esquina inferior
                 // derecha
  var urlTile;

  // obtiene las coordenadas x/y de las esquinas superior
  // izquierda e inferior derecha
  xySupIzq = new GPoint(tile.x * 256, (tile.y + 1) * 256);
  xyInfDer = new GPoint((tile.x + 1) * 256, tile.y * 256);

  // obtiene las coordenadas geográficas de las esquinas superior
  // izquierda e inferior derecha
  geoSupIzq = G_NORMAL_MAP.getProjection().
    fromPixelToLatLng(xySupIzq, zoom);
  geoInfDer = G_NORMAL_MAP.getProjection().
    fromPixelToLatLng(xyInfDer, zoom);

  // genera la URL del "tile"
  urlTile  = this.wmsUrl;
  urlTile += "&REQUEST=GetMap";
  urlTile += "&SERVICE=WMS";
  urlTile += "&VERSION=" + this.wmsVersion;
  urlTile += "&LAYERS=" + this.wmsCapas;
  urlTile += "&STYLES=" + this.wmsEstilos;
  urlTile += "&FORMAT=" + this.wmsFormato;
  urlTile += "&BGCOLOR=" + this.wmsColorFondo;
  urlTile += "&TRANSPARENT=TRUE";
  urlTile += "&SRS=" + this.wmsSrs;
  urlTile += "&BBOX=" + geoSupIzq.x + "," + geoSupIzq.y + "," +
    geoInfDer.x + "," + geoInfDer.y;
  urlTile += "&WIDTH=256";
  urlTile += "&HEIGHT=256";
  urlTile += "&reaspect=false";

  return urlTile;
}
</script>
</head>
<body onload="initialize()" onunload="GUnload()">
<div id="elemento_mapa" style="width: 750px; height: 450px">
</div>
</body>
</html>
```

El resultado será parecido al siguiente:



Ejemplo 3: mapa WMS combinado con el mapa normal de Google Maps

En este ejemplo, hemos creado un mapa personalizado que combina el mapa normal de Google Maps con un mapa geológico a escala 1:50000 procedente del servicio WMS del ICC. Sólo mostramos la función *initialize*, porque el resto del código no cambia respecto al ejemplo n.º 2.

```
function initialize() {
  var mapa;
  var capaGeo;
  var mapaCombi;

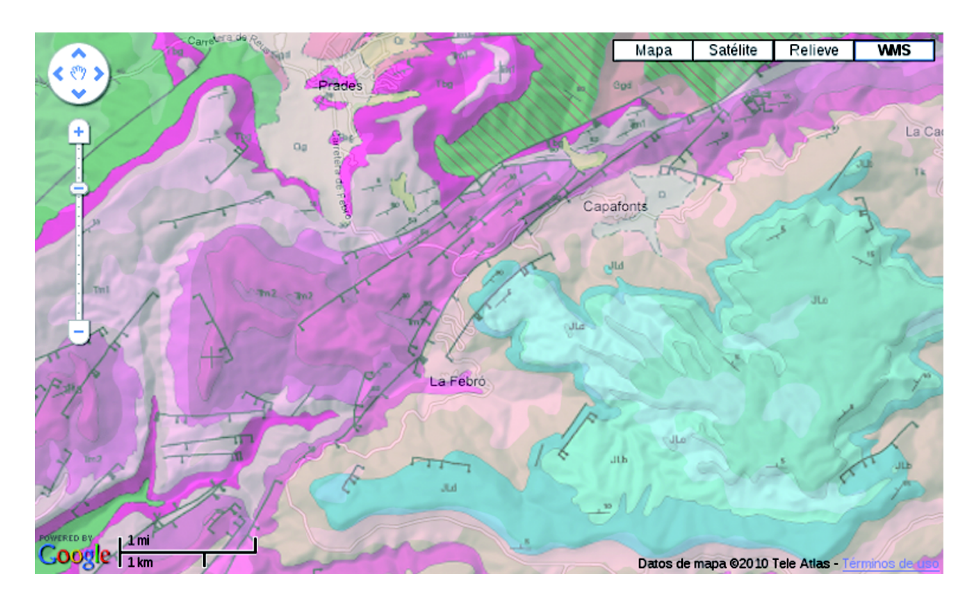
  if (GBrowserIsCompatible()) {
    mapa = new GMap2(document.getElementById("elemento_mapa"));
    mapa.setCenter(new GLatLng(41.5580, 1.5906), 7);
    mapa.setUIToDefault();

    // crea la capa WMS
    capaGeo = crearCapaWms(7, 15, 0.4,
      "http://shagrat.icc.es/lizardtech/iserv/ows?",
      "mgc50m,");

    // crea un mapa personalizado a partir de la capa WMS y del
    // mapa normal de Google Maps
    mapaCombi = new GMapType(
      [G_NORMAL_MAP.getTileLayers()[0], capaGeo],
      G_NORMAL_MAP.getProjection(),
      "WMS");

    // agrega el mapa personalizado al visor
    mapa.addMapType(mapaCombi);
  }
}
```

El resultado será parecido al siguiente:



Ejemplo 4: dos mapas WMS procedentes de una única fuente combinados

El ejemplo siguiente combina dos mapas procedentes del servicio WMS del ICC: el topográfico 1:25000 y el geológico 1:50000. Sólo mostramos la función *initialize*, porque el resto del código no cambia respecto al ejemplo n.º 2.

```
function initialize() {
  var mapa;
  var capaTopo;
  var capaGeo;
  var mapaCombi;

  if (GBrowserIsCompatible()) {
    mapa = new GMap2(document.getElementById("elemento_mapa"));
    mapa.setCenter(new GLatLng(41.5580, 1.5906), 7);
    mapa.setUIToDefault();

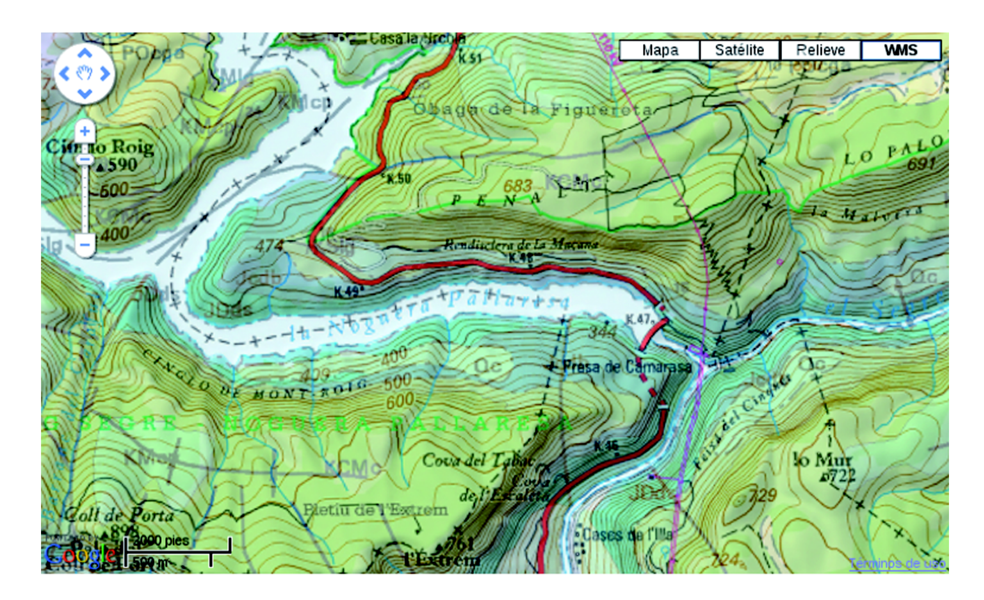
    // crea la capa WMS del mapa topográfico
    capaTopo = crearCapaWms(7, 12, 1.0,
      "http://shagrat.icc.es/lizardtech/iserv/ows?",
      "mtc50m,");

    // crea la capa WMS del mapa geológico
    capaGeo = crearCapaWms(7, 15, 0.40,
      "http://shagrat.icc.es/lizardtech/iserv/ows?",
      "mgc50m,");

    // crea un mapa personalizado a partir de las dos capas WMS
    mapaCombi = new GmapType(
      [capaTopo, capaGeo],
      G_NORMAL_MAP.getProjection(), "WMS");

    // agrega el mapa personalizado al visor
    mapa.addMapType(mapaCombi);
  }
}
```

El resultado será parecido al siguiente:



Ejemplo 5: dos mapas WMS procedentes de fuentes distintas combinados

El ejemplo siguiente combina el mapa topográfico de Cataluña a escala 1:50000 procedente del servicio WMS del ICC con el atlas climático de Cataluña (clima anual, temperatura media) procedente del servicio WMS del CREAF. Sólo mostramos la función *initialize*, porque el resto del código no cambia respecto al ejemplo n.º 2.

```
function initialize() {
  var mapa;
  var capaTopo;
  var capaClima;
  var mapaCombi;
  if (GBrowserIsCompatible()) {
    mapa = new GMap2(document.getElementById("elemento_mapa"));
    mapa.setCenter(new GLatLng(41.5580, 1.5906), 7);
    mapa.setUIToDefault();

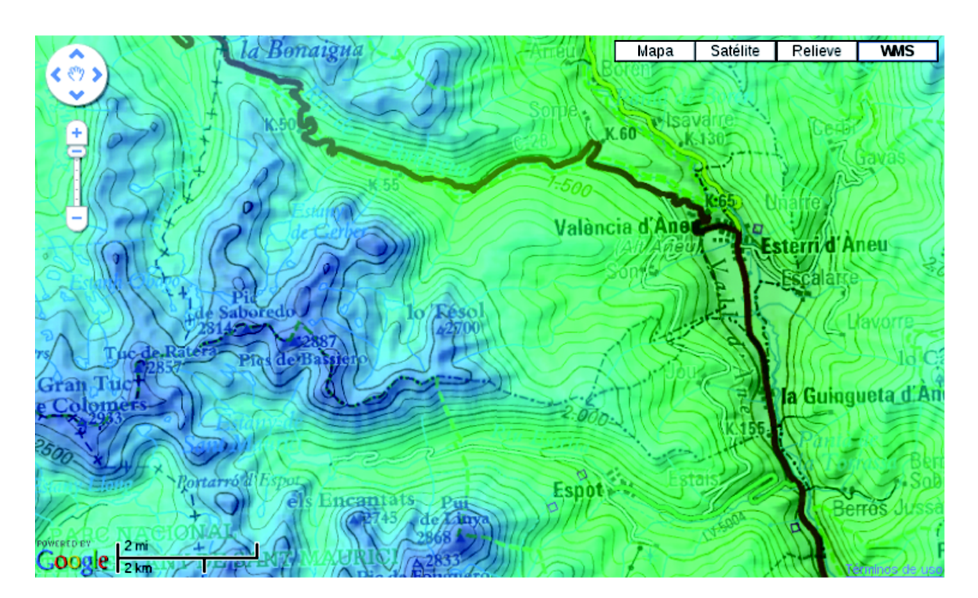
    // crea la capa WMS del mapa del ICC
    capaTopo = crearCapaWms(7, 12, 0.5,
      "http://shagrat.icc.es/lizardtech/iserv/ows?",
      "mtc250m,");

    // crea la capa WMS del mapa del CREAF
    capaClima = crearCapaWms(7, 12, 1.0,
      "http://www.opengis.uab.es/cgi-bin/ACDC/MiraMon5_0.cgi?",
      "clima_anual-catalunya",
      "Tmit",
      "image/jpeg");

    // crea un mapa personalizado a partir de las dos capas WMS
    mapaCombi = new GMapType(
      capaClima, capaTopo],
      G_NORMAL_MAP.getProjection(),
      "WMS");

    // agrega el mapa personalizado al visor
    mapa.addMapType(mapaCombi);
  }
}
```

El resultado será parecido al siguiente:



Resumen

En este módulo hemos aprendido a ir más allá de las capacidades de Google Maps, conectándolo con bases de datos de mapas externas mediante el protocolo OGC/WMS. Esta forma de trabajar nos da acceso a una gran variedad de cartografía temática (muchas veces de carácter local) mucho más precisa que la ofrecida por Google, a la vez que permite el acceso a unos mismos datos a través de diversas fuentes, aumentando así la disponibilidad y la confiabilidad de los datos.

Para poder utilizar mapas procedentes de una fuente WMS en Google Maps, hemos aprendido, en primer lugar, a obtener mapas WMS a través de un navegador web, de la misma forma que accederíamos a cualquier sitio de la red. Una vez hecho esto, para poder visualizarlos, ha sido necesario crear mapas personalizados que posteriormente hemos agregado como una capa en alguno de los tipos de mapa predeterminados de Google Maps (mapa, satélite o relieve), o en uno de nuevo.

Al final, hemos aprendido a combinar mapas de Google Maps, con mapas WMS, e incluso mapas WMS procedentes de diferentes fuentes.

Bibliografía

Institut Cartogràfic de Catalunya, "*Exemple bàsic Google Maps amb capes ICC*", http://www.icc.es/cat/content/download/9942/32328/file/gm_wms_helloworld.zip.

Just van den Broecke, "*Google Maps API Experiments*", http://www.geoskating.com/gmap/.

Carlos Granell Canut, "*Servicios OGC*", FUOC 2009.

Google, "*Google Maps JavaScript API V2 Reference*", http://code.google.com/intl/es/apis/maps/documentation/javascript/v2/reference.html.

Google, "*Google Maps API Concepts*", http://code.google.com/intl/es/apis/maps/documentation/javascript/v2/basics.html.

Open GIS Consortium Inc., "*Web Map Service Implementation Specification*", http://portal.opengeospatial.org/files/?artifact_id=1081&version=1&format=pdf, version 1.1.1, 16/01/2002.